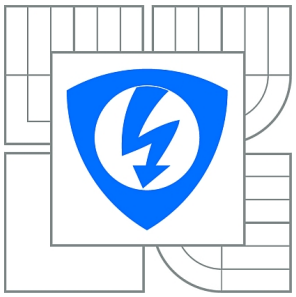
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ
ÚSTAV TELEKOMUNIKACÍ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION
DEPARTMENT OF TELECOMMUNICATIONS

ZABEZPEČENÍ VYSOKORYCHLOSTNÍCH KOMUNIKAČNÍCH SYSTÉMŮ

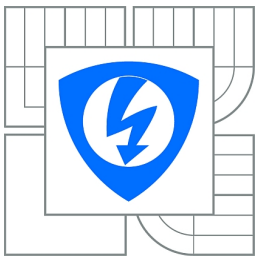
PROTECTION OF HIGHSPEED COMMUNICATION SYSTEMS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — Bc. DAVID SMÉKAL
AUTHOR

VEDOUCÍ PRÁCE — Ing. JAN HAJNÝ, Ph.D.
SUPERVISOR

BRNO 2015



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Fakulta elektrotechniky a komunikačních technologií

Ústav telekomunikací

Diplomová práce

magisterský navazující studijní obor
Telekomunikační a informační technika

Student: Bc. David Smékal ***ID:*** 134405
Ročník: 2 ***Akademický rok:*** 2014/2015

NÁZEV TÉMATU:

Zabezpečení vysokorychlostních komunikačních systémů

POKYNY PRO VYPRACOVÁNÍ:

V rámci projektu se student seznámí se síťovými kartami založenými na platformě FPGA a základy šifrovacích systémů. Cílem projektu je nastudovat proces vývoje na platformě FPGA, naprogramovat jednoduché operace manipulace s pakety, navrhnout strukturu šifrování provozu pomocí AES a vytvořit software pro šifrování provozu na FPGA. Výstupem bude funkční software pro šifrování dat šifrou AES v módu ECB, případně CBC, pomocí FPGA.

DOPORUČENÁ LITERATURA:

[1] STALLINGS, William. Cryptography and network security: principles and practice. Seventh edition. xix, 731 pages. ISBN 01-333-5469-5.
[2] FIPS-197. ADVANCED ENCRYPTION STANDARD (AES). USA: NIST, 2001. Dostupné z: http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf
[3] INVEA-TECH NetCOPE. [online]. [cit. 2014-10-07]. Dostupné z: https://www.invea.com/cs/produkty-sluzby/fpga-development-kit/netcope

Termín zadání: 9.2.2015 ***Termín odevzdání:*** 26.5.2015

Vedoucí práce: Ing. Jan Hajný, Ph.D.
Konzultanti diplomové práce:

doc. Ing. Jiří Mišurec, CSc.
Předseda oborové rady

UPOZORNĚNÍ:

ABSTRAKT

Diplomová práce se zabývá šifrováním dat pomocí AES a jejich implementací pomocí jazyka VHDL na síťovou FPGA kartu. V teoretické části práce je vysvětlen algorimus šifrování AES, jeho jednotlivé kroky a použité operační módy. Dále je popsán programovací jazyk VHDL, jeho vývojové prostředí Vivado, FPGA karty a konfigurovatelný framework NetCope. Praktickou částí práce je implementace šifry AES–128 v jazyce VHDL, jejíž výstup byl použit v FPGA kartě, která vykoná šifrování. Pomocí simulace byly efektivně odladěny chyby a dále bylo možné provést syntézu. Toto vše bylo prováděno za pomoci vývojového softwaru Vivado. Posledním krokem praktické části práce bylo testování na kartě COMBO-80G. Na FPGA kartu byly implementovány celkem 4 projekty. Dva z nich jsou šifrování a dešifrování ECB módu AES algoritmu a zbylé dva popisují šifrování a dešifrování módu CBC.

KLÍČOVÁ SLOVA

Šifrování, dešifrování, AES, VHDL, FPGA, NetCOPE, Vivado, ECB, CBC, Combo-80G, XOR, šifra, bloková šifra, AddRoundKey, SubBytes, ShiftRows, MixColumns, substituční tabulka, mixovací matice, klíč, implementace, simulace, iterace, testování, matice, framework, firmware, FrameLinkUnaligned, MI32, algoritmus, násobení

ABSTRACT

The diploma thesis deals with 128–bit AES data encryption and its implementation in FPGA network card using VHDL programming language. The theoretical part explains AES encryption and decryption, its individual steps and operating modes. Further was described the VHDL programming language, development environment Vivado, FPGA network card Combo–80G and configurable framework NetCOPE. The practical part is the implementation of AES–128 in VHDL. A simulation was used to eliminate errors, then the synthesis was performed. These steps were made using Vivado software. Last step of practical part was testing of synthesized firmware on COMBO–80G card. Total of 4 projects were implemented in FPGA card. Two of them were AES encryption and decryption with ECB mode and another two describe the encryption and decryption with CBC mode.

KEYWORDS

Encryption, decryption, AES, VHDL, FPGA, NetCOPE, Vivado, ECB, CBC, Combo-80G, XOR, cipher, block cipher, AddRoundKey, SubBytes, ShiftRows, MixColumns, substitution table, mixing matrix, key, implementation, simulation, iteration, test, matrix, framework, firmware, FrameLinkUnaligned, MI32, algorithm, multiply

SMÉKAL, David *Zabezpečení vysokorychlostních komunikačních systémů*: diplomová práce. Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav telekomunikací, 2015. 55 s. Vedoucí práce byl Ing. Jan Hajný, Ph.D.

PROHLÁŠENÍ

Prohlašuji, že svou diplomovou práci na téma „Zabezpečení vysokorychlostních komunikačních systémů“ jsem vypracoval samostatně pod vedením vedoucího diplomové práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Jako autor uvedené diplomové práce dále prohlašuji, že v souvislosti s vytvořením této diplomové práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a/nebo majetkových a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon), ve znění pozdějších předpisů, včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č. 40/2009 Sb.

Brno ....

(podpis autora)

PODĚKOVÁNÍ

Rád bych poděkoval svému vedoucímu diplomové práce panu Ing. Janu Hajnému, Ph.D. za odborné vedení a konzultace, za cenné poznámky a připomínky k obsahu i zpracování. Dále děkuji odbornému konzultantovi diplomové práce Ing. Denisi Matouškovi za pomoc a poskytnuté rady při tvorbě práce. Především za trpělivost a čas strávený při množství konzultací.

Brno ....

(podpis autora)

Faculty of Electrical Engineering
and Communication
Brno University of Technology
Purkynova 118, CZ-61200 Brno
Czech Republic

http://www.six.feec.vutbr.cz

PODĚKOVÁNÍ

Výzkum popsaný v této diplomové práci byl realizován v laboratořích podpořených z projektu SIX; registrační číslo CZ.1.05/2.1.00/03.0072, operační program Výzkum a vývoj pro inovace.

Brno . . . . . .

. . . . . .
(podpis autora)

OBSAH

SEZNAM OBRÁZKŮ

SEZNAM TABULEK

ÚVOD

Diplomová práce se věnuje problematice zabezpečení vysokorychlostních komunikačních systémů. Je zde obsažen přehled šifrovacích metod a především představení vybraného algoritmu, který se dnes řadí mezi nejrozšířenější. Jedná se o šifrovací algoritmus AES (anglicky Advanced Encryption Standard). Cílem je implementace tohoto algoritmu na platformě síťových karet FPGA. Jde o programovatelná hradlová pole (anglicky Field Programmable Gate Array), která umožňují vývoj hardwarově akcelerovaných aplikací. Bližší popis je uveden v následujících kapitolách.

Úkolem práce je tedy vhodně implementovat zmíněný AES šifrovací algoritmus v jazyce VHDL. Navrhnout funkční software pro šifrování dat šifrou AES v módu ECB, případně CBC.

Samotná práce je rozdělena do několika částí:

- **Šifrování** – obecné informace o kryptografii, blokovém šifrování a bližší popis AES algoritmu a jeho provozních režimech.
- **Úvod do VHDL, FPGA a NetCOPE** – seznámení s programovacím jazykem, vývojovým prostředím, programovatelnými logickými obvody a především se síťovou kartou Combo–80G.
- **Implementace šifry AES ve VHDL** – popis praktické části, implementace základních bloků algoritmu, ukázka návrhu systému.
- **Testování** – zohlednění dosažených výsledků v diplomové práci, popis ovládání šifrování.

1 ŠIFROVÁNÍ

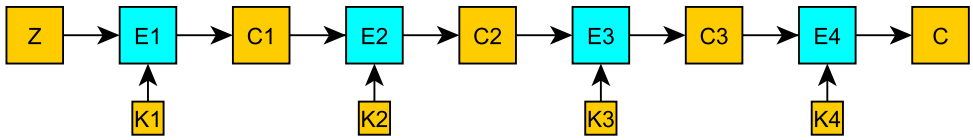
Kryptografie je věda, která se věnuje utajováním informace převodem do neznámé podoby, která je čitelná jen za určitých znalostí. Podle [2] se kryptografie zabývá vytvářením složitých matematických metod pro zajištění bezpečnosti zpráv. Za zprávu rozumíme posloupnost symbolů, která nese potřebnou informaci. Šifrovací algoritmus (šifra) slouží k transformaci informace do nesrozumitelné podoby a naopak. K tomu je potřeba číselný parametr, který se nazývá šifrovací klíč. Od šifrovací transformace je požadováno, aby se bez znalosti dešifrovacího klíče nedala odvodit originální zpráva z šifrového textu.

Symetrické šifrování označujeme tehdy, jestliže se používají na obou stranách systému klíče, které jsou odvoditelné jeden od druhého a ve většině případů se jedná o stejné klíče na každé straně. Důležité je utajení klíče před neoprávněnou osobou. Symetrický kryptosystém může být dvojího druhu. Proudová šifra nebo bloková šifra. Práce je zaměřena pouze na blokovou šifru, která provádí šifrování po blocích.

Blokové šifry pracují s bloky zprávy pevné velikosti. Šifrování se provádí po blocích o pevné délce `n` bitů. Ke každému vstupnímu bloku `Z` je přiřazen tajný klíč `K`, kde na výstupu dostaneme blok šifrového textu `C` dle rovnice

$$C = E(Z, K), \tag{1.1}$$

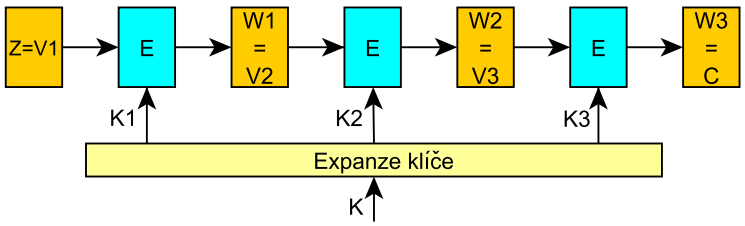
funkce `E` je postavena tak, že hodnota bloku `C` závisí na všech bitech vstupního bloku `Z` a na všech bitech klíče `K`. Většina blokových šifer se nevyužívá samostatně (pouze jednou), ale vícekrát za sebou, kdy docílíme tzv. kaskádové šifry. Výstupní blok jedné šifry je vstupem následující šifry a výsledný blok poslední šifry je výsledný šifrový text (viz obrázek 1.1).



Obr. 1.1: Popis kaskádové šifry

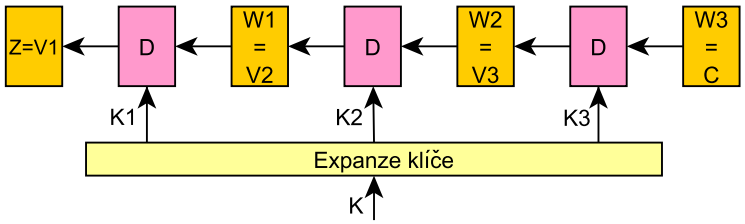
Nejčastěji se používá tzv. iterovaná šifra, kdy v bloku expanze se z šifrovacího klíče `K` odvodí více dílčích klíčů. Tyto klíče jsou šifrovacími klíči pro opakovaně použitou iteraci `E`. Iterace je jednoduchá šifra, ve které je vstupní blok bitů `V` v závislosti na iteračním klíči `K` transformován vhodnou kombinací blokových operací (substitucí, permutací a aritmetických operací) do podoby výstupního bloku `W` [2]. Výstupem

poslední iterace `E` je výstupní blok `C`, což je výsledný šifrovaný text. Obecné schéma blokové šifry při šifrování je následující.



Obr. 1.2: Obecné schéma blokvé šifry při šifrování

Dešifrování je u blokové šifry prováděno analogicky jako u šifrování. Transformace `D` je inverzní vůči transformaci `E` a inverzní je i pořadí použití iteračních klíčů `K` [2]. Obecné schéma blokové šifry při dešifrování je zřejmé z obrázku 1.3.



Obr. 1.3: Obecné schéma blokové šifry při dešifrování

1.1 Bloková šifra AES

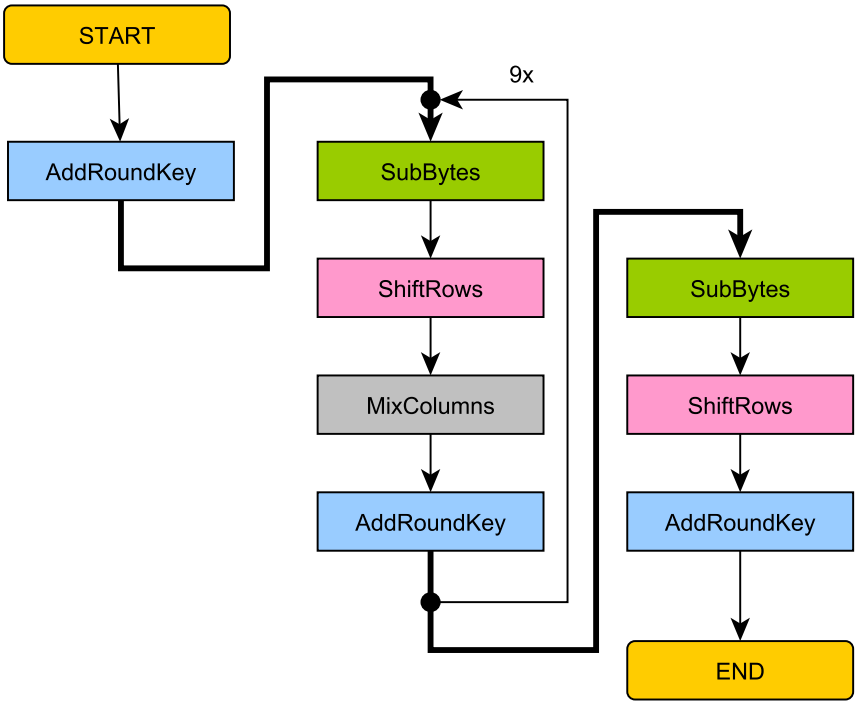
Advanced Encryption Standard, neboli AES, je algoritmus používaný k šifrování dat. Jedná se o symetrickou blokovou šifru, která zpracovává data rozdělená do bloků pevně dané délky 128 bitů. Tyto bity jsou uspořádány do matice 4×4, kdy jedna buňka matice odpovídá jednomu bajtu. K šifrování a dešifrování se používá stejný klíč o velikosti 128, 192 nebo 256 bitů. Šifrování probíhá v tzv. `iteracích`, jejichž počet závisí na délce klíče. Pro každou iteraci se odvozují klíče ze šifrovacího klíče, proces nazývaný expanze klíče. Algoritmus lze rozdělit na tyto tři dílčí šifry:

- **iniciační část**
  - Key Expansion – expanze klíče,
  - AddRoundKey – každý bajt stavové matice je zkombinován s klíčem za pomoci operace XOR,
- **hlavní šifra, iterace**
  - SubBytes – každý bajt je nahrazen jiným podle vyhledávací tabulky,
  - ShiftRows – každý řádek matice je postupně posunut o určitý počet kroků,
  - MixColumns – násobení matice „mixovací“ maticí,
  - AddRoundKey – přičtení iteračního klíče,
- **závěrečná část**
  - SubBytes – substituce bajtů,
  - ShiftRows – rotace řádků,
  - AddRoundKey – sečtení stavové matice s klíčem.

Na začátku šifrování se provede expanze klíče. Ke stavové matici 4×4 vytvořené z bloku 128bitů (8×16 bajtů) se v šifře přičte klíč. Poté se devětkrát provede hlavní šifra. Počet provedení hlavní šifry se odvíjí v závislosti na použití délky klíče. Číslo 9 odpovídá klíči 128 bitů. Iterace se skládá ze substituce bajtů stavové matice (SubBytes), rotace řádků (ShiftRows), následně substituce sloupců (MixColumns). Na konci každé hlavní šifry se k matici přičte iterační klíč (AddRoundKey). V závěrečné části se opět uskuteční substituce bajtů, rotace řádků a poslední fází je přičtení klíče finální šifry. Ve výsledné matici jsou uloženy bajty kryptogramu. Tyto popsané kroky jsou zobrazeny ve vývojovém diagramu 1.4

1.1.1 SubBytes – substituce bajtů

Jedná se o prostou substituci, kde každému vstupnímu bajtu je přiřazena předem daná hodnota výstupního bajtu. Přiřazení se provádí dle známé tabulky 1.1. Každý bajt se rozdělí na dvě hexadecimální číslice. Pomocí první se určí řádek a pomocí druhé sloupec v tabulce. V dané buňce se pak přečte substituovaný bajt.



Obr. 1.4: Jednotlivé kroky šifrování AES

1.1.2 ShiftRows – rotace řádků

Při rotaci řádků se upravují jednotlivé řádky matice následujícím způsobem. V prvním řádku matice se neprovede řádná změna. Ve druhém řádku se provede rotace vlevo o jeden bajt, ve třetím řádku rotace vlevo o dva bajty a ve čtvrtém řádku se provede rotace vlevo o tři bajty.

1.1.3 MixColumns – substituce sloupců

Transformace MixColumns vychází z vynásobení získané matice tzv. mixovací maticí 1.2. Sloupec původní matice se násobí mixovací maticí, a vznikne tak sloupec nové transformované matice.

Násobení jedničkou znamená ponechání původní hodnoty. Násobení dvojkou znamená posuv původní hodnoty o jeden bit vlevo. Násobení trojkou znamená posuv původní hodnoty o jeden bit vlevo a následné sečtení (XOR) s původní hodnotou. Pokud je výsledek vyšší než 255, pak se k němu ještě přičte hodnota 11B v hexadecimálním tvaru.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
3	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
4	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
5	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
6	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
7	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
8	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
A	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
B	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
C	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
D	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
E	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
F	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

Tab. 1.1: Substituční tabulka pro transformaci SubBytes

2	3	1	1
1	2	3	1
1	1	2	3
3	1	1	2

Tab. 1.2: Mixovací matice

2	3	1	1		a		2a + 3b + 1c + 1d
1	2	3	1	×	b	=	1a + 2b + 3c + 1d
1	1	2	3		c		1a + 1b + 2c + 3d
3	1	1	2		d		3a + 1b + 1c + 2d

Tab. 1.3: Použití mixovací matice pro MixColumns

1.1.4 AddRoundKey – přičtení iteračního klíče

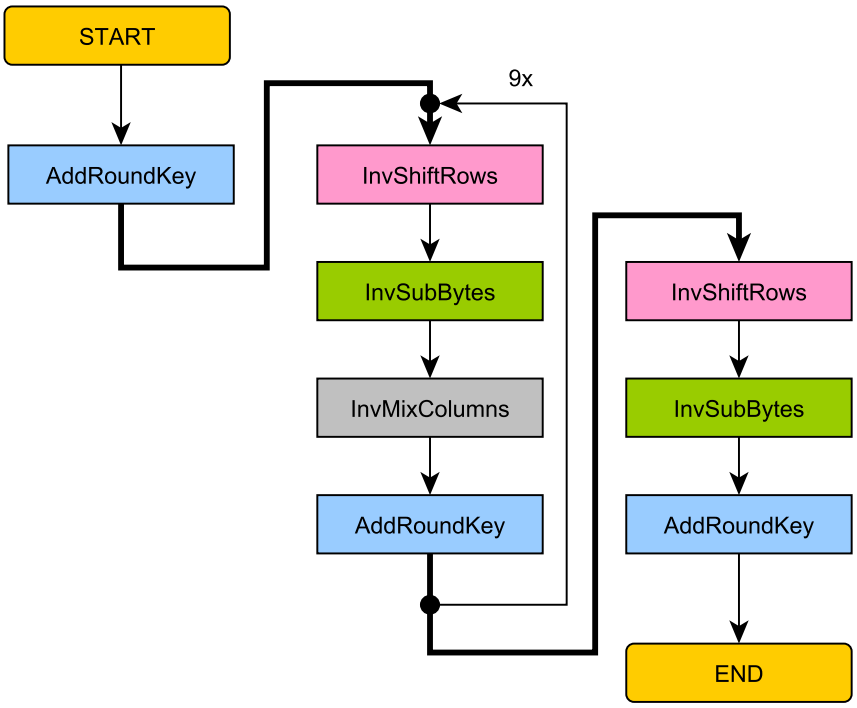
Přičtení iteračního klíče je poslední transformací. Je třeba provést expanzi klíče, kdy se z obecného klíče K odvozují dílčí klíče o stejném formátu jako původní matice, tj. 4×4. Následná operace sečtení matice a klíče je prováděna pomocí exklusivního logického součtu XOR.

1.2 Dešifrování AES

Dešifrování je opačný postup šifrování, kde výstupem bude informace, která byla na počátku celého procesu. Dešifrovací algoritmus popisuje krok za krokem, jak s blokem dat nakládat, abychom obdrželi správnou informaci. Provádí se v inverzním pořadí za použití inverzních transformací.

Inverzní transformací se rozumí operace, která je blízká původní změně, ale je v opačném pořadí nebo používá jiné hodnoty. K šifrovací transformaci SubBytes je inverzní transformací InvSubBytes. Dále k ShiftRows je inverzní transformací InvShiftRows, k MixColumns je to InvMixColumns. Poslední transformace AddRoundKey je inverzní sama k sobě, jedná se pouze o operaci XOR, ale i tak můžeme transformaci označit jako InvAddRoundKey.

Podle [2] je při dešifrování oproti šifrování potřeba dvou změn v prohození pořadí. Konkrétně jde o prohození prvních dvou transformací (SubBytes a ShiftRows) a druhých dvou transformací (MixColumns a AddRoundKey). Při dešifrování se používají transformace v pořadí InvShiftRows, InvSubBytes, AddRoundKey, InvMixColumns. Tím je zajištěno korektní dešifrování.



Obr. 1.5: Jednotlivé kroky dešifrování AES

V následujícím textu si představíme čtyři základní inverzní transformace.

1.2.1 InvShiftRows

U transformace InvShiftRows se provádí rotace řádků opačným směrem oproti ShiftRows. První řádek matice se ponechá beze změny. Ve druhém řádku se provede rotace vpravo o jeden bajt, ve třetím řádku rotace vpravo o dva bajty a ve čtvrtém řádku se provede rotace vpravo o tři bajty.

1.2.2 InvSubBytes

Transformace InvSubBytes provádí substituci pomocí příslušné tabulky 1.4 stejným způsobem, jako tomu je u transformace SubBytes popsané v kap. 1.1.1.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	52	09	6a	d5	30	36	a5	38	bf	40	a3	9e	81	f3	d7	fb
1	7c	e3	39	82	9b	2f	ff	87	34	8e	43	44	c4	de	e9	cb
2	54	7b	94	32	a6	c2	23	3d	ee	4c	95	0b	42	fa	c3	4e
3	08	2e	a1	66	28	d9	24	b2	76	5b	a2	49	6d	8b	d1	25
4	72	f8	f6	64	86	68	98	16	d4	a4	5c	cc	5d	65	b6	92
5	6c	70	48	50	fd	ed	b9	da	5e	15	46	57	a7	8d	9d	84
6	90	d8	ab	00	8c	bc	d3	0a	f7	e4	58	05	b8	b3	45	06
7	d0	2c	1e	8f	ca	3f	0f	02	c1	af	bd	03	01	13	8a	6b
8	3a	91	11	41	4f	67	dc	ea	97	f2	cf	ce	f0	b4	e6	73
9	96	ac	74	22	e7	ad	35	85	e2	f9	37	e8	1c	75	df	6e
A	47	f1	1a	71	1d	29	c5	89	6f	b7	62	0e	aa	18	be	1b
B	fc	56	3e	4b	c6	d2	79	20	9a	db	c0	fe	78	cd	5a	f4
C	1f	dd	a8	33	88	07	c7	31	b1	12	10	59	27	80	ec	5f
D	60	51	7f	a9	19	b5	4a	0d	2d	e5	7a	9f	93	c9	9c	ef
E	a0	e0	3b	4d	ae	2a	f5	b0	c8	eb	bb	3c	83	53	99	61
F	17	2b	04	7e	ba	77	d6	26	e1	69	14	63	55	21	0c	7d

Tab. 1.4: Substituční tabulka pro transformaci InvSubBytes

1.2.3 InvMixColumns

Transformace InvMixColumns je inverzní k původní operaci MixColumns. Jde o podobný výpočet nových bajtů matice jako při šifrování, ale používá se jiná mixovací matice. Při šifrování matice nabívá hodnot 1, 2, a 3, u dešifrování jsou tyto hodnoty vyšší. Využívá se veličina 9, 11, 13 a 14. Přesné uspořádání inverzní mixovací matice je uvedeno v 1.5. Místo složitého výpočtu násobení původních bajtů a hodnot

14	11	13	9
9	14	11	13
13	9	14	11
11	13	9	14

Tab. 1.5: Inverzní mixovací matice

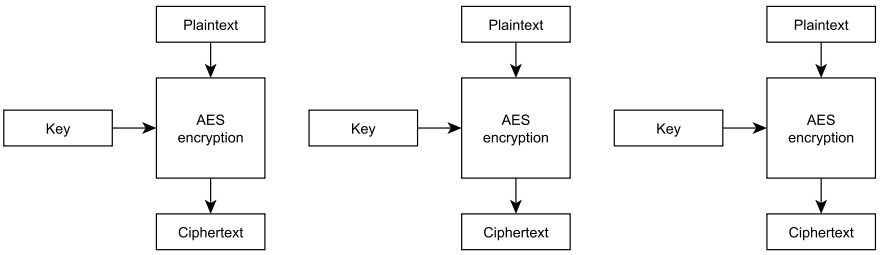
z mixovací matice lze implementovat předem vypočtené tabulky, ve kterých stačí vyhledat správný bajt a určit z něj novou hodnotu. Předem vypočtená vyhledávací tabulka obsahuje všechny možné kombinace bajtů. V našem případě je to 255 různých hodnot, které jsou vynásobeny potřebnou hodnotou (9, 11, 13, 14). Musíme mít tedy 4 tabulky, které lze uložit do návrhu FPGA. Tento návrh je více popsán kapitole 1.1.3. Jak takovéto uspořádání může vypadat prezentují tabulky 3.3, 3.4, 3.5, 3.6.

1.3 Provozní operační módy blokových šifer

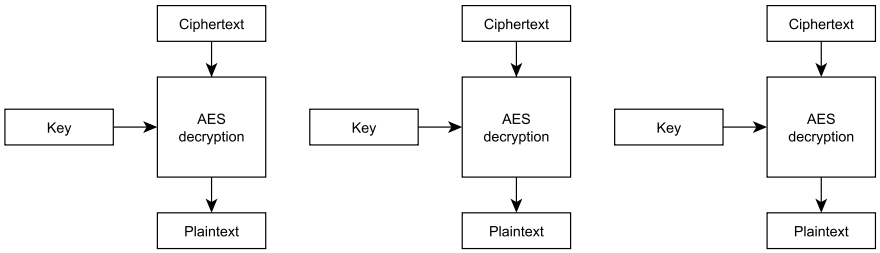
Pro bezpečnější užití blokové šifry v různých aplikacích byly navrženy různé režimy, které modifikují standard o další operace. Vylepšují tím vlastnosti blokových šifer pro danou aplikaci. Nejčastěji jsou to režimy označované zkratkami ECB, CBC, CTR, CFB, OFB a XTS. My se zaměříme pouze na první dva.

1.3.1 Operační mód ECB

Režim ECB (Electronic Codebook) pojednává o šifrování bloků nezávisle na sobě. Postupně zpracuje jednotlivé bloky, které šifruje nezávisle na ostatních stejným klíčem. Bloky následně skládá za sebe a vytváří tak šifrový text. Dešifrování odpovídá opačnému pořadí operací. Výhodou ECB módu je rychlost a jednoduchost. Je vhodný pro šifrování krátkých zpráv. Značnou nevýhodou podle [2] je skutečnost, že kdyby zpráva obsahovala stejné bloky dat, výstup bude opět stejný a útočník může tohoto jevu využít. Schéma módu ECB je zobrazeno na 1.6 pro šifrování a na 1.7 pro dešifrování.



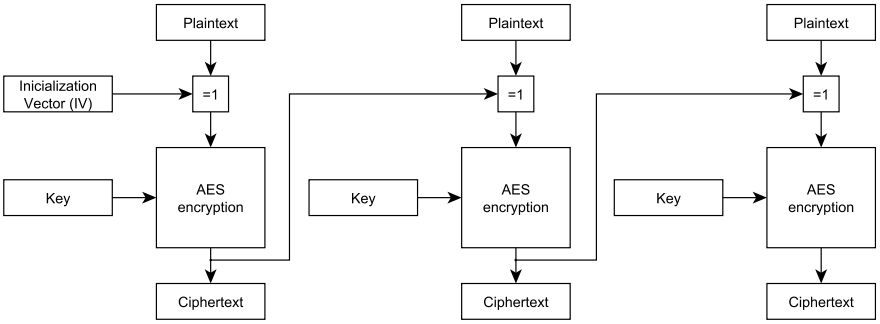
Obr. 1.6: Schéma šifrování v módu ECB



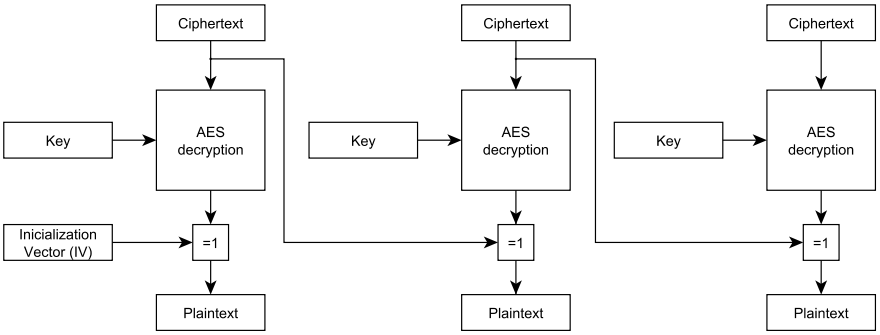
Obr. 1.7: Schéma dešifrování v módu ECB

1.3.2 Operační mód CBC

Režim CBC (Cipher Block Chaining) je bezpečnější než předchozí mód, jelikož jsou bloky na sebe závislé. Jestliže budeme šifrovat dva stejné bloky, výsledek bude vždy jiný. Blok zprávy je před samotným šifrováním „xorován“ s předchozím šifrovým textem a teprve nově vzniklý blok je zašifrován. Pro šifrování prvního bloku je nutné použít unikátní, předem známý vstupní blok o stejné délce jako je blok dat (128 bitů), který se označuje jako inicializační vektor. Tento inicializační vektor je tajný, neodhadnutelný a znají ho obě strany. Bližší popis uvádí schéma šifrování 1.8 a dešifrování 1.9.



Obr. 1.8: Schéma šifrování v módu CBC



Obr. 1.9: Schéma dešifrování v módu CBC

2 ÚVOD DO VHDL, FPGA A NETCOPE

V této kapitole se seznámíme s potřebnými znalostmi ohledně návrhu a vývoje systému, který lze implementovat na fyzické zařízení. Je třeba specifikovat programovací jazyk, ve kterém budeme obvod programovat. Dále vývojové prostředí, které použijeme pro snadný návrh, simulaci a syntézu. Vysvětlíme si, co jsou programovatelné logické obvody, proč využijeme právě FPFA karty a jaké konkrétní typy.

2.1 Programovací jazyk – VHDL

Jazyk VHDL (anglicky Very High Speed Integrated Circuit Hardware Description Language) je jazyk navržený speciálně pro popis, návrh a simulaci číslicových systémů a obvodů [1]. Je jedním z nejpoužívanějších jazyků pro popis hardwarových struktur hradlových polí. Má rozsáhlé vyjadřovací schopnosti a velkou univerzálnost použití pro různé číslicové systémy. Velkou výhodou je možnost provádět návrhy pro hradlová pole různých výrobců. Výsledná implementace závisí pouze na kompilaci VHDL kódu pro konkrétní zařízení.

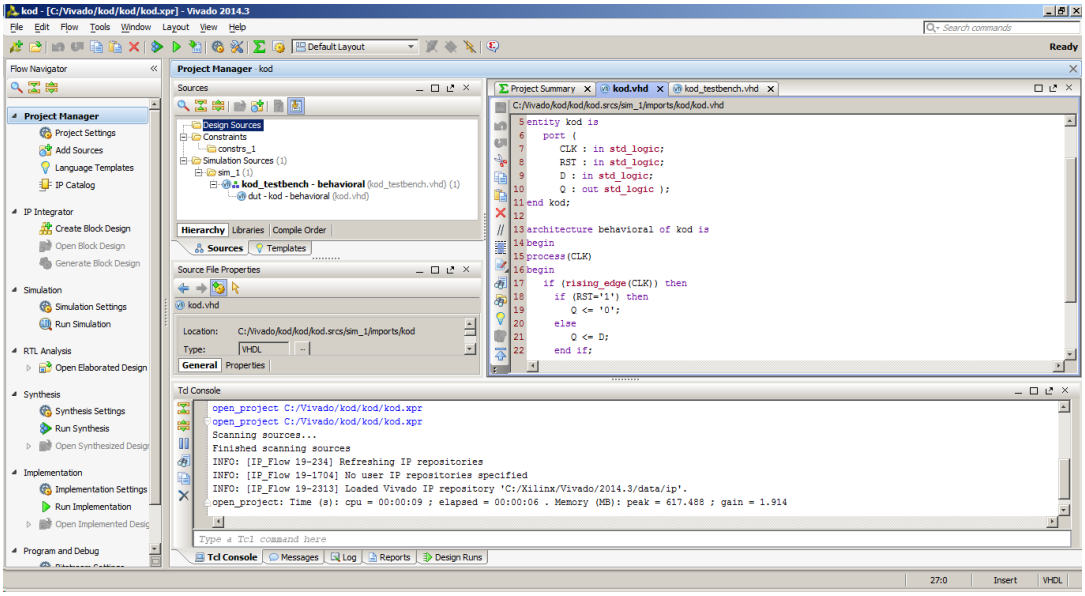
Od klasických programovacích jazyků se liší v popisu číslicového systému, který bude v konečné fázi realizován na hardware [1]. Vytvořený kód je třeba nechat projít syntézou, jejímž výsledkem je zapojení hradel a klopných obvodů určených pro programovatelné logické obvody, tedy koncové FPGA zařízení. Návrh struktury lze přirovnat k propojování skutečných fyzických obvodů. Samotné bloky návrhu jsou ve finálním modulu propojeny pomocí objektu typu signál, který odpovídá reálnému elektrickému signálu.

Součástí práce není popisovat programovací jazyk, ale je nezbytné uvést pár základních údajů. Jazyk VHDL má dvě povinné komponenty, které jsou nezbytné v každém návrhu. První z návrhových jednotek je tzv. entita. Entita definuje vstupní a výstupní signály objektu, přiřazuje typ a směr přenosu dat. Takovýto objekt se může tvářit jako logické hradlo nebo celý obvod. Druhou povinnou jednotkou je komponenta architektura. Ta definuje chování a funkci entity. Pracuje s porty entity a vytváří vlastní logické funkce obvodu. Každá entita musí mít alespoň jednu architekturu.

Architektura může obsahovat více příkazů, kdy se příkazy budou vykonávat současně nezávisle na sobě. Takový popis architektury může být různého typu. Častěji se hovoří o stylu popisu, kdy se může jednat o styl behaviorální, o styl popisující tok dat (DATA-FLOW) a o styl strukturální.

2.2 Vývojové prostředí

Již bylo zmíněno, že existuje více výrobců zařízení a s tím i související vývojová prostředí, především vhodné kompilátory. Pro vytváření uceleného projektu je zapotřebí zvolit správný software pro zvýšení produktivity programátora. Jde o zaměření na jeden konkrétní programovací jazyk, tedy VHDL. Práce je tvořena v prostředí Vivado® od společnosti Xilinx. Prostředí je vyobrazeno na snímku 2.1.

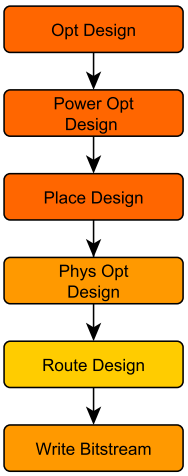


Obr. 2.1: Vývojové prostředí Vivado®

Implementace ve Vivado se skládá z logických a fyzických návrhů. Jde o šest kroků, které zajistí převod ze zdrojového kódu programu na konfigurační soubor pro programovatelný logický obvod. Tyto kroky jsou zobrazeny v diagramu 2.2. Implementace lze rozdělit na dvě fáze. Tyto fáze jsou často označovány jako „Place and Route“, kdy lze pojem chápat jako umístění a směrování.

- Opt Design – Optimalizuje log. návrh a dělá jej jednodušší pro cílové zařízení.
- Power Opt Design – Optimalizuje návrhové prvky ke snížení výkonových požadavků na zařízení.
- Place Design – Umístí návrh na cílové zařízení.
- Phys Opt Design – Optimalizuje časování
- Route Design – Tvoří cesty návrhu na cílovém zařízení.
- Write Bitstream – Generuje bitový tok pro konečnou konfiguraci.

První čtyři kroky rozhodují o tom, kam umístit abstraktní elektronické součástky a logické prvky. Pátý krok má na starost propojení vodičů k připojení umístěné komponenty. Poslední fází je generování konečného bitového toku, jako výstup celé implementace.



Obr. 2.2: Vivado implementace

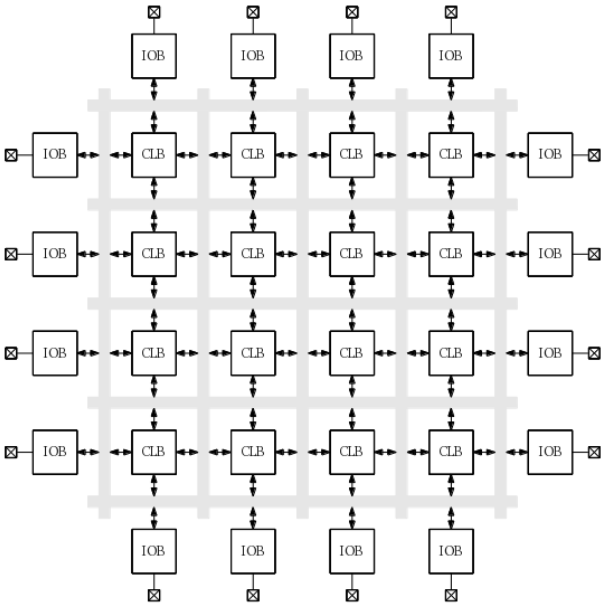
Vývoj softwaru probíhá na vzdáleném laboratorním serveru, s operačním systémem CentOS a vývojovým protředím Vivado. Připojování probíhá přes SSH Tunel na vzdálený počítač. Je využit program pro vzdálenou plochu VNC (Virtual Network Computing), aby práce v grafickém prostředí operačního systému byla snažší. Po připojení přes tunel na laboratorní síť je možnost přistupovat k síťové FPGA kartě, na které bude šifrování aplikováno.

2.3 FPGA karty

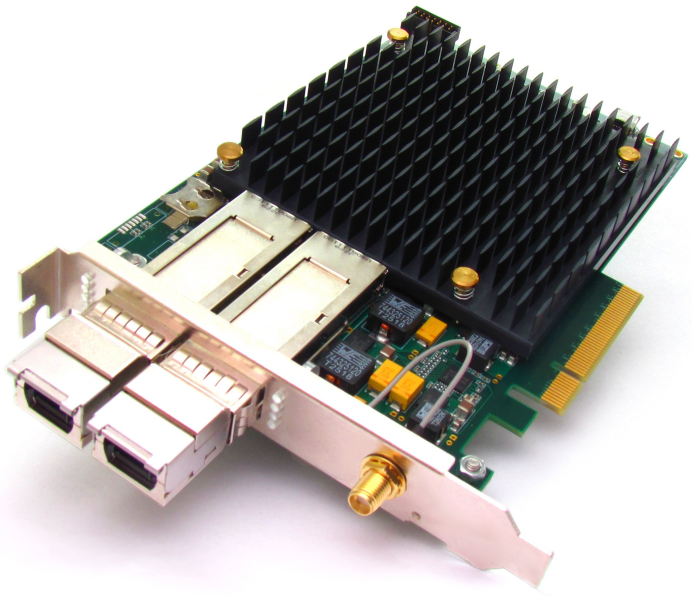
FPGA (anglicky Field Programmable Gate Array) jsou programovatelné logické obvody, které lze naprogramovat až po jejich výrobě. Místo vývoje zařízení pro konkrétní funkci, se zvolená funkce nastaví po výrobě a lze ji i průběžně měnit v závislosti na požadavcích konkrétního zadání. Výhodou těchto logických obvodů je jejich univerzálnost.

Ze všech různých logických programovatelných obvodů jsou FPGA nejobecnější obvody a obsahují nejvíce logiky. FPGA karta obsahuje spoustu bloků, které jsou mezi sebou propojeny a vytváří tak abstraktní propojovací matici. Obvody mohou být vstupně–výstupní nebo logické. Vstupně-výstupní bloky obvykle obsahují registr, budič, multiplexer a ochranné obvody. Logické bloky naopak představují vlastní programovatelné logické jednotky. Nejpoužívanější struktura konfigurovatelného logického bloku je znázorněna na obrázku 2.3, kde `IOB` (Input/Output Block) představují vstupně-výstupní bloky a `CLB` (Configurable Logic Block) programovatelné logické bloky.

Při praktické realizaci návrhu zabezpečení využíváme síťovou FPGA kartu COMBO-



Obr. 2.3: Struktura obvodu typu FPGA [6]



Obr. 2.4: Karta COMBO-80G [4]

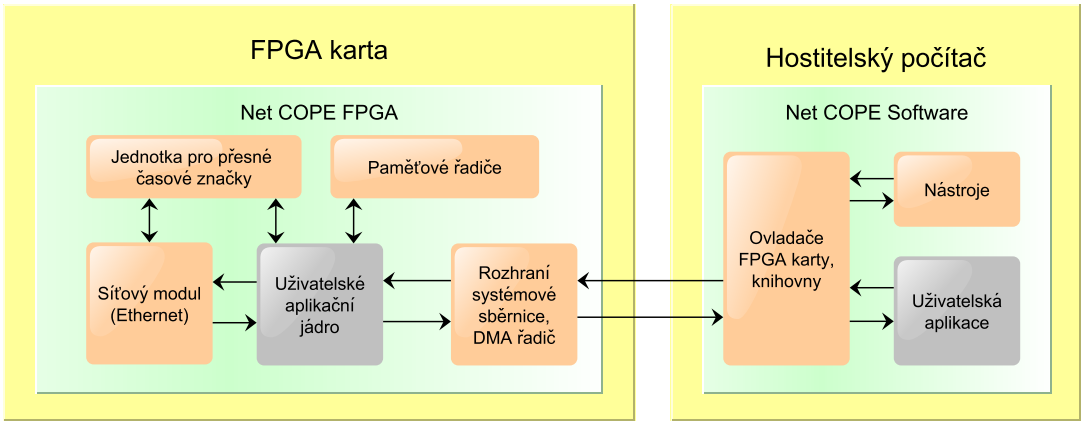
80G 2.4 od společnosti Invea-Tech, umožňující vývoj hardwarově akcelerovaných aplikací [4]. Karta je osazena výkonným FPGA čipem Virtex–7 firmy Xilinx. Podporuje technologie 40G a 10G Ethernet. Fyzické rozhraní je realizováno pomocí dvou konektorů QSFP+ s možností použití monomódu, multimódu, CWDM nebo metalických transceiverů. V případě QSFP+ transceiverů osazených na kartě Combo

jde o převod optických signálů na elektrické a naopak. Karta využívá sběrnici PCI Express pro vysokorychlostní přenosy dat mezi kartou a hostitelským počítačem s propustností 50 Gb/s.

2.4 NetCOPE

NetCOPE je konfigurovatelný framework od společnosti INVEA-TECH, který vývojáři umožní snadný vývoj aplikací na FPGA kartách. Obecně je cílem frameworku snadnější vývoj aplikace tak, aby se návrháři a vývojáři mohli zaměřit na svůj úkol. Framework tvoří nezávislou vrstvu mezi hardwarem a uživatelským softwarem. Poskytuje tak jednoduché rozhraní pro hardwarové aplikace. Jsou v něm implementované řadiče a rozhraní pro práci s periferiemi karet Combo.

Díky NetCOPE mohou vytvořené aplikace v FPGA komunikovat se softwarovou aplikací běžící na hostitelském počítači. Umožňuje odesílat a přijímat data ze softwaru na FPGA kartu a obsluhovat ji z koncového zařízení, např. z počítače. Framework NetCOPE je rozdělen na dvě části. První částí je myšlen firmware určený pro FPGA kartu a druhou částí je software navržený pro hostitelským počítač. Komunikace mezi firmwarem na síťové kartě Combo a softwarovou aplikací běžící na hostitelském počítači probíhá ve frameworku NetCOPE přes rozhraní PCI Express. Struktura NetCOPE je znázorněna na snímku 2.5.



Obr. 2.5: Struktura NetCOPE [5]

2.4.1 Komunikace v NetCOPE

Komunikace v samotné kartě Combo je řízena různými protokoly, my si nyní představíme dva hlavní, se kterými při návrhu šifrování pracujeme. Dále si vysvětlíme,

jakým způsobem je usnadněna komunikace mezi hostitelským počítačem a samotnou kartou.

Protokol FrameLinkUnaligned

Komunikační protokol FrameLinkUnaligned, zkráceně FLU, vychází z protokolu FrameLink, který popisuje vysokorychlostní přenosy dat. Rozhraní FLU definuje sadu signálů, díky kterým je možné přenášet rámce libovolného formátu. Je určen pro sběrnice 256 bitů a více.

Oproti FrameLink se vyznačuje v řadě výhod. Především podporuje nezarovnaný začátku paketu – kdy paket začíná se určí pomocí SOP_POS signálu. Dále využívá pozitivní logiku, kdy logická jednička odpovídá vysoké úrovni a naopak. Všechny signály jsou synchronní na hodinový signál CLK.

V následující tabulce 2.1 je seznam signálů protokolu FLU:

CLK	Clock	hodinový signál
DATA	Data	přenášená data
SOP	Start of packet	určuje začátek paketu v aktuálním hodinovém taktu
SOP_POS	Start of packet position	oznamuje začátek části přenášeného paketu
EOP	End of packet	určuje platný konec paketu současného datového slova
EOP_POS	End of packet position	oznamuje, kde se nachází poslední paket v datovém slově
SRC_RDY	Source ready	vysílač je připraven vysílat data
DST_RDY	Destination ready	přijímač je připraven přijímat data

Tab. 2.1: Signály protokolu FrameLinkUnaligned

Protokol MI32

Protokol MI32 slouží pro nízkoúrovňovou komunikaci s pevně danou šířkou datového slova 32 bitů. Je určen pro řídící operace, nikoliv pro vysokorychlostní datové přenosy jako je tomu u FLU. Podporuje přenosy jak pro čtení, tak i pro zápis. Všechny signály jsou synchronní na hodinový signál CLK. Pro uživatelské aplikace má MI32 vyhrazen adresový prostor pro své signály a registry. Tento prostor je definovaný pevnou adresou začínající na 0x0200 0000_h a končící na adrese 0x03FF $FFFF_h$.

V následující tabulce 2.2 je seznam signálu protokolu MI32:

Zápis lze provést následovně. Je nutné využít signál MI32_WR. Jakmile se objeví data pro zápis na MI32_DWR, je třeba znát adresu, kam se data uloží pomocí MI32_ADDR a v neposlední řadě je třeba zápis potvrdit signálem MI32_BE. U čtení je to podobné, využívá se signál MI32_RD. Adresa, ze které se bude číst je dána signálem MI32_ADDR a potvrzením MI32_BE. V tomto kroku signál MI32_ARDY potvrzuje správnost adresy a data se objeví na MI32_DRD.

MI32_DWR	Data to be written	data pro zápis
MI32_DRD	Data to be read	data pro čtení
MI32_ADDR	Address	určuje pracovní adresu
MI32_RD	Read	určuje, že se jedná o čtení
MI32_WR	Write	určuje, že se jedná o zápis
MI32_BE	Byte Enable	povolení dat na zápis
MI32_ARDY	Address ready	adresa je k dispozici
MI32_DRDY	Data ready	data jsou připravená

Tab. 2.2: Signály protokolu MI32

Softwarová aplikace

Aby obsluha karty byla uživatelsky jednodušší, lze vytvořit program v jazyku C, který bude komunikovat s aplikačním jádrem samotné karty. Samotný vývoj se realizuje programováním v jazyku C, následnou komplilací programu do spustitelné podoby a spuštěním programu, který vykoná definované instrukce. Kompilace i spuštění programu se provádí na serveru. Program obsahuje volaní hotových připravených funkcí z knihoven, které jsou implementovány přímo v NetCOPE. Bližší význam a syntaxi funkcí lze dohledat v interní dokumentaci NetCOPE pod označením „Software Doxygen documentation“. Díky přichystaným knihovnám, je jazyk C pro tento návrh nejvhodnější. Druhou možností by bylo vytvořit bash skript, který by vykonával podobné operace. V praktické části si takový program v C představíme a vysvětlíme jakým způsobem komunikuje s kartou a čím nám usnadní práci.

3 IMPLEMENTACE ŠIFRY AES VE VHDL

Vlastní práce spočívá v implementaci šifrovacího algoritmu AES ve VHDL jazyce. Následný výstup bude použit v FPGA síťové kartě, která bude vykonávat šifrování. Tato práce pojednává pouze o AES–128, tedy použití 128 bitového klíče. Co je to AES a jak pracuje je uvedeno v kapitole 1.1. V této části se zaměříme především na 4 základní bloky, které je třeba implementovat v jazyce VHDL. Jak již bylo řečeno, jedná se o bloky SubBytes, ShiftRows, MixColumns, AddRoundKey. Nyní si představíme způsob implementace jednotlivých transformací.

3.1 SubBytes

Vstupní blok dat, tedy 16 bajtů, rozdělíme na jednotlivé bajty, se kterými následně pracujeme. Lépe řečeno porovnáváme aktuální bajt se substituční tabulkou, kterou máme definovanou jako výběrové přiřazení. Deklarace přiřazení se skládá ze všech 256 možných kombinací vstupního bajtu. Následující kód vysvětluje zápis substituce s několika bajty z tabulky.

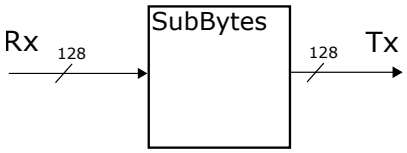
```
architecture behavior of SubTable is
begin
  with inp select
    outp <= x"63" when x"00",
            x"7c" when x"01",
            x"77" when x"02",
            ...
            x"54" when x"FD",
            x"BB" when x"FE",
            x"16" when others;
end architecture;
```

V samotné entitě nahrazujeme zjištěný bajt hodnotou ze SubTable. Pakliže vstupní blok dat je 16 bajtů, každý bajt se nahradí novou hodnotou určenou z výše uvedeného seznamu.

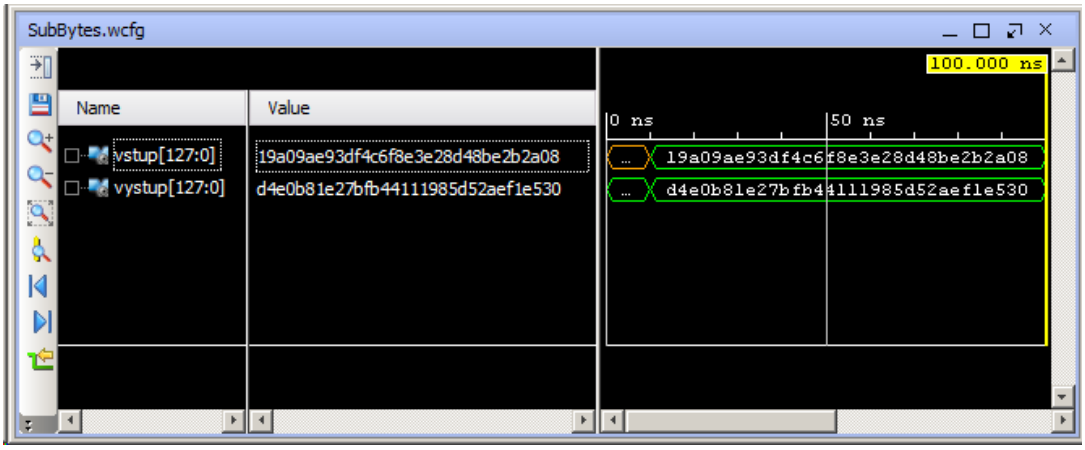
```
gen: for i~in 0 to 15 generate
sub : entity work.SubTable port map(
        inp => vstup(8*i+7 downto 8*i),
        outp => vystup(8*i+7 downto 8*i));
end generate gen;
```

Pro příklad je možné použít vstupní posloupnost `19 A0 9A E9 3D F4 C6 F8 E3 E2 8D 48 BE 2B 2A 08` a na výstupu po substituci je očekáván výsledek `D4`

`E0 B8 1E 27 BF B4 41 11 98 5D 52 AE F1 E5 30`. Blokové schéma 3.1 znázorňuje vstupní a výstupní signál, dále na snímku 3.2 je zachycen výstup z prostředí Xilinx Vivado.



Obr. 3.1: Blokové schéma SubBytes

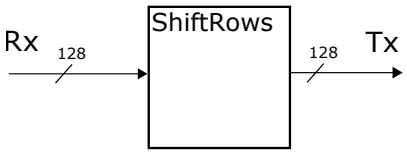


Obr. 3.2: Praktická ukázka SubBytes

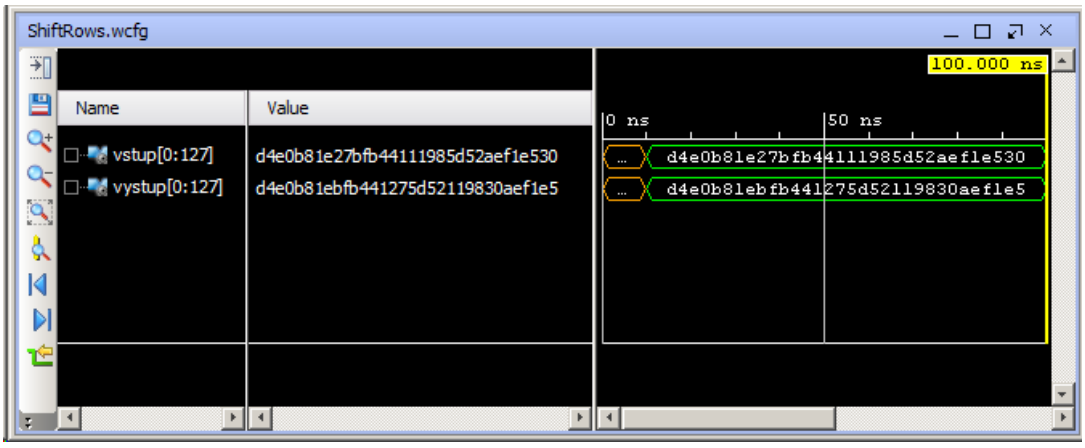
3.2 ShiftRows

V druhém bloku se pracuje s jednotlivými řádky matice a provádí se u nich posun bajtů o určitou pozici. Je důležité si uvědomit, že matici reprezentujeme vektorem. Je třeba mít na paměti, že první 4 bajty tvoří první řádek, další 4 bajty druhý řádek atd. Pracujeme proto s indexem jednotlivých bitů, které transformujeme dle teoretických pravidel. Přesný postup je následující.

```
architecture structural of ShiftRows is
begin
    TX(127 downto 96) <= RX(127 downto 96);
    TX(95 downto 64)  <= RX(87 downto 64) & RX(95 downto 88);
    TX(63 downto 32)  <= RX(47 downto 32) & RX(63 downto 48);
    TX(31 downto 0)   <= RX(7 downto 0)   & RX(31 downto 8);
end structural;
```



Obr. 3.3: Blokové schéma ShiftRows

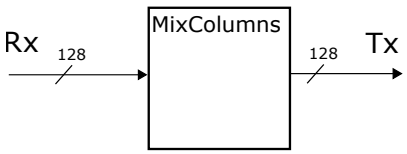


Obr. 3.4: Praktická ukázka ShiftRows

Pro ukázku opět snímek 3.4, kde je zachycen výstup v prostředí Xilinx Vivado. Na výstupu bloku SubBytes jsme obdrželi `D4 E0 B8 1E 27 BF B4 41 11 98 5D 52 AE F1 E5 30`, což je nyní vstupem komponenty ShiftRows. Jednotivé kroky jsou zobrazeny na vývojovém diagramu 1.4. Po transformaci všech řádků dostáváme na výstupu `D4 E0 B8 1E BF B4 41 27 5D 52 11 98 30 AE F1 E5`.

3.3 MixColumns

Tato část algoritmu je v praktické části rozdělena na 3 vrstvy, kdy v každé je určitá VHDL komponenta, která se stará o danou část výpočtu. Násobení bajtů má na starost komponenta `MultiplyX`, kde „X“ značí hodnotu, kterou chceme násobit. Vytvoření sloupců a násobení mixovací maticí provádí komponenta `MultiplyColumns` a v posledním kroku komponenta `MixColumns` uspořádá bajty do správného formátu, kdy na výstupu dostaneme 16 bajtů transformovaných dat.



Obr. 3.5: Blokové schéma MixColumns

Multiply

Do první vrtvy jsou zahrnuty samotné operace s bajty. Jedná se o násobení bajtu určitou hodnotou, jak je uvedeno v teoretické části 1.1.3. Vstup se násobí číslem 2 a 3, resp. 9, 11, 13, 14 u dešifrování. Způsoby, jak výpočet vyřešit, jsou dva.

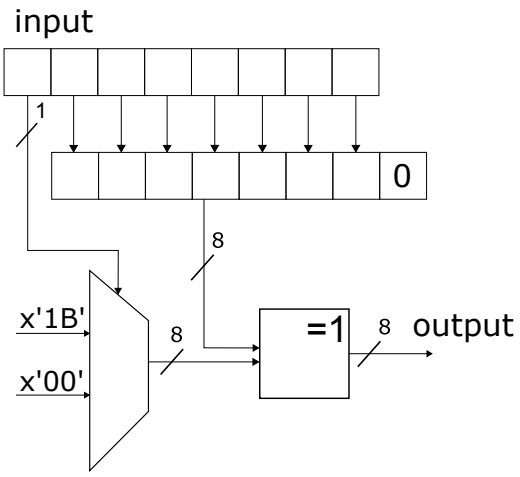
První je zcela logický. Bajt vynásobíme požadovanou hodnotou a uložíme. Operace není zdaleka tak jednoduchá. Jestliže vynásobíme bajt o velikosti větší než 127 ($7F_h$) dvěma resp. bajt o velikosti větší než 85 (55_h) třemi, vznikne nám číslo o devíti bitech a nemůžeme využít výstupní vektor, který je velký 8 bitů. V tomto kroku je třeba využít teoretické informace podle [2] a bajt „xorovat“ s $11B_h$. Tato hodnota je zjištěna výpočtem a díky logické operaci XOR se výsledek vždy dostane pod hranici 255 bitů a odpovídá správnému výsledku v Galoisově poli.

Druhá možnost, je zcela odlišná. Implementaci transformace MixColumns je možné vyřešit způsobem podobný tomu, jak je řešena transformace SubBytes. Tedy substitucí vstupních dat dle předem známých hodnot. Vstupní bajt může nabývat 256 různých hodnot. Jestliže všechny tyto kombinace předem vynásobíme hodnotou, kterou chceme znát na výstupu, můžeme vytvořit substituční tabulku, podle které přiřadíme výstupu správnou hodnotu.

V samotném návrhu se jedná o entitu MultiplyTwo, jestliže se bavíme o násobení dvěma. Ta obsahuje dvě architektury, které provádí totéž. Avšak zcela odlišně a je jen na nás, jaký způsob výpočtu použijeme. První z nich je architektura `logic`, která odpovídá logickému výpočtu popsanému výše. Výpis zdrojového kódu je následující.

```
architecture logic of MultiplyTwo is
begin
   output2 <= input2(6 downto 0)&'0' when input2(7)='0'
              else input2(6 downto 0)&'0' xor x"1B";
end logic;
```

V první řadě nás zajímá, zda při násobení překročíme hodnotu, kdy by došlo k přetečení o jeden bit. Jestliže ano, s výsledkem je nutné provést operaci XOR s hodnotou $1B_h$. Násobení dvěma se provede posunutím bitových hodnot o jednu pozici vlevo a přidá se logická nula. Pro lepší pochopení je třeba uvést schéma návrhu 3.6, které vystihuje výše popsané kroky i uvedený zdrojový kód.



Obr. 3.6: Schéma realizace násobení dvěma (logic)

Druhá architektura řešící další možnost výpočtu násobení je `lookup`. Ta obsahuje vyhledávací tabulku, ve které jsou všechny předem vypočítané hodnoty po násobení dvěma. V ukázkovém zdrojovém kódu je možné vidět, jakým způsobem jsou hodnoty interpretovány. Vstupem entity je bajtový signál input2, kterému se přiřadí nová hodnota a uloží se do signálu output2.

```
architecture lookup of MultiplyTwo is
begin
   with input2 select
      output2 <= x"00" when x"00",
                 x"02" when x"01",
                 x"04" when x"02",
                   ...
                 x"E1" when x"FD",
                 x"E7" when x"FE",
                 x"E5" when others;
end lookup ;
```

Tabulka pro násobení dvěma může vypadat následovně 3.1. V první buňce tabulky je výsledek násobení $00_h \times 2 = 00$, v druhé buňce $01_h \times 2 = 02$, ..., v předposlední buňce je výsledek $FE_h \times 2 = E7$ a v poslední buňce je výsledek $FF_h \times 2 = E5$.

00	02	04	06	08	0a	0c	0e	10	12	14	16	18	1a	1c	1e
20	22	24	26	28	2a	2c	2e	30	32	34	36	38	3a	3c	3e
40	42	44	46	48	4a	4c	4e	50	52	54	56	58	5a	5c	5e
60	62	64	66	68	6a	6c	6e	70	72	74	76	78	7a	7c	7e
80	82	84	86	88	8a	8c	8e	90	92	94	96	98	9a	9c	9e
a0	a2	a4	a6	a8	aa	ac	ae	b0	b2	b4	b6	b8	ba	bc	be
c0	c2	c4	c6	c8	ca	cc	ce	d0	d2	d4	d6	d8	da	dc	de
e0	e2	e4	e6	e8	ea	ec	ee	f0	f2	f4	f6	f8	fa	fc	fe
1b	19	1f	1d	13	11	17	15	0b	09	0f	0d	03	01	07	05
3b	39	3f	3d	33	31	37	35	2b	29	2f	2d	23	21	27	25
5b	59	5f	5d	53	51	57	55	4b	49	4f	4d	43	41	47	45
7b	79	7f	7d	73	71	77	75	6b	69	6f	6d	63	61	67	65
9b	99	9f	9d	93	91	97	95	8b	89	8f	8d	83	81	87	85
bb	b9	bf	bd	b3	b1	b7	b5	ab	a9	af	ad	a3	a1	a7	a5
db	d9	df	dd	d3	d1	d7	d5	cb	c9	cf	cd	c3	c1	c7	c5
fb	f9	ff	fd	f3	f1	f7	f5	eb	e9	ef	ed	e3	e1	e7	e5

Tab. 3.1: Vypočítané hodnoty násobení dvěma (lookup)

U násobení třemi je to podobné, pouze se uvažují jiné operace s bajtem. Násobení třemi provedeme posuvem hodnoty o jeden bit doleva a přičtením původní hodnoty operací exklusivního logického součtu. Tento postup vystihuje blokové schéma 3.7 s ukázkou zdrojového kódu. Násobení třemi řeší entita MultiplyThree.

```
architecture logic of MultiplyThree is
begin
   output3 <= input3(6 downto 0)&'0' xor input3(7 downto 0)
    when input3(7)='0'
    else input3(6 downto 0)&'0' xor input3(7 downto 0) xor x"1B";
end logic;
```

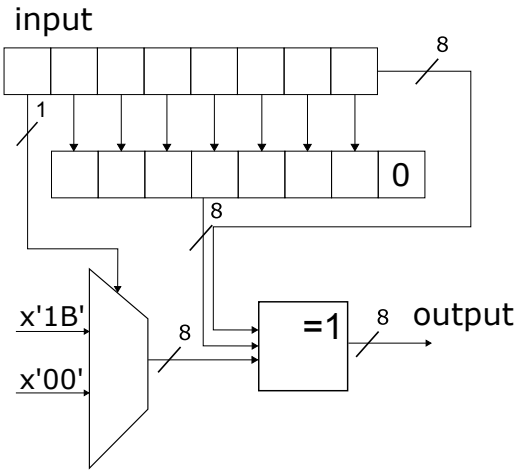
Obdobně je tomu u architektury lookup, kdy si do tabulky 3.2 předem vypočítáme všechny možné kombinace a opět přiřazujeme výstupu správnou hodnotu.

```
architecture lookup of MultiplyThree is
begin
   with input3 select
      output3 <= x"00" when x"00",
                 x"03" when x"01",
```

```
                    x"06" when x"02",
                    ...
                    x"1C" when x"FD",
                    x"19" when x"FE",
                    x"1A" when others;
end lookup ;
```

00	03	06	05	0c	0f	0a	09	18	1b	1e	1d	14	17	12	11
30	33	36	35	3c	3f	3a	39	28	2b	2e	2d	24	27	22	21
60	63	66	65	6c	6f	6a	69	78	7b	7e	7d	74	77	72	71
50	53	56	55	5c	5f	5a	59	48	4b	4e	4d	44	47	42	41
c0	c3	c6	c5	cc	cf	ca	c9	d8	db	de	dd	d4	d7	d2	d1
f0	f3	f6	f5	fc	ff	fa	f9	e8	eb	ee	ed	e4	e7	e2	e1
a0	a3	a6	a5	ac	af	aa	a9	b8	bb	be	bd	b4	b7	b2	b1
90	93	96	95	9c	9f	9a	99	88	8b	8e	8d	84	87	82	81
9b	98	9d	9e	97	94	91	92	83	80	85	86	8f	8c	89	8a
ab	a8	ad	ae	a7	a4	a1	a2	b3	b0	b5	b6	bf	bc	b9	ba
fb	f8	fd	fe	f7	f4	f1	f2	e3	e0	e5	e6	ef	ec	e9	ea
cb	c8	cd	ce	c7	c4	c1	c2	d3	d0	d5	d6	df	dc	d9	da
5b	58	5d	5e	57	54	51	52	43	40	45	46	4f	4c	49	4a
6b	68	6d	6e	67	64	61	62	73	70	75	76	7f	7c	79	7a
3b	38	3d	3e	37	34	31	32	23	20	25	26	2f	2c	29	2a
0b	08	0d	0e	07	04	01	02	13	10	15	16	1f	1c	19	1a

Tab. 3.2: Vypočítané hodnoty násobení třemi (lookup)



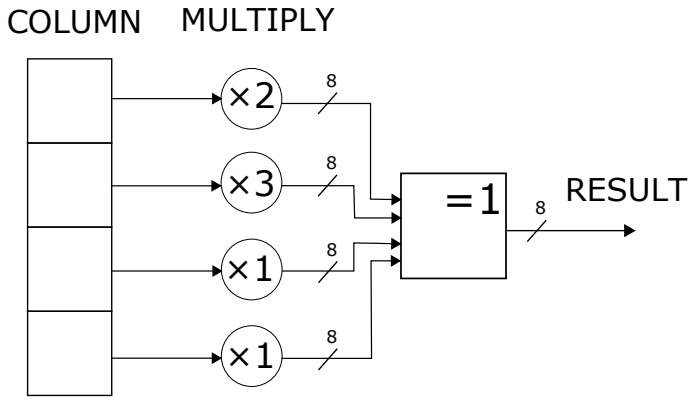
Obr. 3.7: Schéma realizace násobení třemi (logic)

MultiplyColumns

V této části výpočtu entita pracuje s jedním vstupním signálem (column) o velikosti 32 bitů, reprezentující sloupec původní matice a jedním výstupním signálem (result), který vrací hodnotu nového bajtu. V ukázkovém příkladu si představíme výpočet prvního bajtu nové matice. V tabulce 1.3 se jedná o první řádek, kdy násobíme první bajt dvojkou, druhý bajt trojkou a třetí i čtvrtý bajt jsou ponechány beze změny. Výsledek je získán součtem nových bajtů. Zdrojový kód i blokové schéma 3.8 je zobrazeno níže.

Jednotlivé bajty, které je třeba vynásobit jsou předány entitě MultiplyTwo, resp. MultiplyThree. Výsledkem je signál output2 (resp. output3). Nový bajt vznikne exklusivním součtem mezi čtyřmi bajty, jenž reprezentuje signál result.

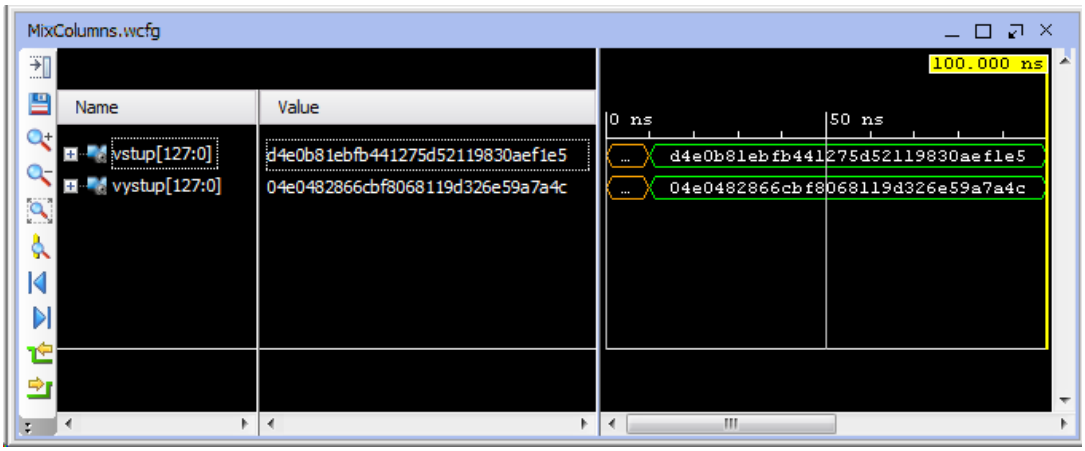
```
input2 => column(31 downto 24),    output2 => output2 );
input3 => column(23 downto 16),    output3 => output3 );

result <= output2 xor output3 xor column(15 to 8) xor column(7 to 0);
```



Obr. 3.8: Blokové schéma MultiplyColumns

MixColumns

V poslední části jsou instancovány předchozí entity. Těmto entitám jsou předávány vstupní hodnoty prezentující sloupce matice a výstup předchozí entity result je přiveden na správnou pozici nové matice. Tím je dokončena celá transformace MixColumns. Na výstupu bloku ShiftRows jsme obdrželi `D4 E0 B8 1E BF B4 41 27 5D 52 11 98 30 AE F1 E5`, což je nyní vstupem komponenty MixColumns. Po transformaci dostáváme `04 E0 48 28 66 CB F8 06 81 19 D3 26 E5 9A 7A 4C`, jak je uvedeno na snímku 3.9.



Obr. 3.9: Praktická ukázka MixColumns

Dešifrování

Aby návrh MixColumns byl kompletní, je třeba uvést i hodnoty pro dešifrování, které jsou uloženy v tabulkách 3.3, 3.4, 3.5, 3.6.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	00	09	12	1b	24	2d	36	3f	48	41	5a	53	6c	65	7e	77
1	90	99	82	8b	b4	bd	a6	af	d8	d1	ca	c3	fc	f5	ee	e7
2	3b	32	29	20	1f	16	0d	04	73	7a	61	68	57	5e	45	4c
3	ab	a2	b9	b0	8f	86	9d	94	e3	ea	f1	f8	c7	ce	d5	dc
4	76	7f	64	6d	52	5b	40	49	3e	37	2c	25	1a	13	08	01
5	e6	ef	f4	fd	c2	cb	d0	d9	ae	a7	bc	b5	8a	83	98	91
6	4d	44	5f	56	69	60	7b	72	05	0c	17	1e	21	28	33	3a
7	dd	d4	cf	c6	f9	f0	eb	e2	95	9c	87	8e	b1	b8	a3	aa
8	ec	e5	fe	f7	c8	c1	da	d3	a4	ad	b6	bf	80	89	92	9b
9	7c	75	6e	67	58	51	4a	43	34	3d	26	2f	10	19	02	0b
A	d7	de	5	cc	f3	fa	e1	e8	9f	96	8d	84	bb	b2	a9	a0
B	47	4e	55	5c	63	6a	71	78	0f	06	1d	14	2b	22	39	30
C	9a	93	88	81	be	b7	ac	a5	d2	db	c0	c9	f6	ff	e4	ed
D	0a	03	18	11	2e	27	3c	35	42	4b	50	59	66	6f	74	7d
E	a1	a8	b3	ba	85	8c	97	9e	e9	e0	fb	f2	cd	c4	df	d6
F	31	38	23	2a	15	1c	07	0e	79	70	6b	62	5d	54	4f	46

Tab. 3.3: Vyhledávací tabulka pro násobení 9

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	00	0b	16	1d	2c	27	3a	31	58	53	4e	45	74	7f	62	69
1	b0	bb	a6	ad	9c	97	8a	81	e8	e3	fe	f5	c4	cf	d2	d9
2	7b	70	6d	66	57	5c	41	4a	23	28	35	3e	0f	04	19	12
3	cb	c0	dd	d6	e7	ec	f1	fa	93	98	85	8e	bf	b4	a9	a2
4	f6	fd	e0	eb	da	d1	cc	c7	ae	a5	b8	b3	82	89	94	9f
5	46	4d	50	5b	6a	61	7c	77	1e	15	08	03	32	39	24	2f
6	8d	86	9b	90	a1	aa	b7	bc	d5	de	c3	c8	f9	f2	ef	e4
7	3d	36	2b	20	11	1a	07	0c	65	6e	73	78	49	42	5f	54
8	f7	fc	e1	ea	db	d0	cd	c6	af	a4	b9	b2	83	88	95	9e
9	47	4c	51	5a	6b	60	7d	76	1f	14	09	02	33	38	25	2e
A	8c	87	9a	91	a0	ab	b6	bd	d4	df	c2	c9	f8	f3	ee	e5
B	3c	37	2a	21	10	1b	06	0d	64	6f	72	79	48	43	5e	55
C	01	0a	17	1c	2d	26	3b	30	59	52	4f	44	75	7e	63	68
D	b1	ba	a7	ac	9d	96	8b	80	e9	e2	ff	f4	c5	ce	d3	d8
E	7a	71	6c	67	56	5d	40	4b	22	29	34	3f	0e	05	18	13
F	ca	c1	dc	d7	e6	ed	f0	fb	92	99	84	8f	be	b5	a8	a3

Tab. 3.4: Vyhledávací tabulka pro násobení 11

0	00	0d	1a	17	34	39	2e	23	68	65	72	7f	5c	51	46	4b
1	d0	dd	ca	c7	e4	e9	fe	f3	b8	b5	a2	af	8c	81	96	9b
2	bb	b6	a1	ac	8f	82	95	98	d3	de	c9	c4	e7	ea	fd	f0
3	6b	66	71	7c	5f	52	45	48	03	0e	19	14	37	3a	2d	20
4	6d	60	77	7a	59	54	43	4e	05	08	1f	12	31	3c	2b	26
5	bd	b0	a7	aa	89	84	93	9e	d5	d8	cf	c2	e1	ec	fb	f6
6	d6	db	cc	c1	e2	ef	f8	f5	be	b3	a4	a9	8a	87	90	9d
7	06	0b	1c	11	32	3f	28	25	6e	63	74	79	5a	57	40	4d
8	da	d7	c0	cd	ee	e3	f4	f9	b2	bf	a8	a5	86	8b	9c	91
9	0a	07	10	1d	3e	33	24	29	62	6f	78	75	56	5b	4c	41
A	61	6c	7b	76	55	58	4f	42	09	04	13	1e	3d	30	27	2a
B	b1	bc	ab	a6	85	88	9f	92	d9	d4	c3	ce	ed	e0	f7	fa
C	b7	ba	ad	a0	83	8e	99	94	df	d2	c5	c8	eb	e6	f1	fc
D	67	6a	7d	70	53	5e	49	44	0f	02	15	18	3b	36	21	2c
E	0c	01	16	1b	38	35	22	2f	64	69	7e	73	50	5d	4a	47
F	dc	d1	c6	cb	e8	e5	f2	ff	b4	b9	ae	a3	80	8d	9a	97

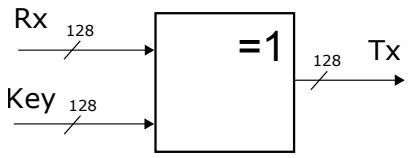
Tab. 3.5: Vyhledávací tabulka pro násobení 13

0	00	0e	1c	12	38	36	24	2a	70	7e	6c	62	48	46	54	5a
1	e0	ee	fc	f2	d8	d6	c4	ca	90	9e	8c	82	a8	a6	b4	ba
2	db	d5	c7	c9	e3	ed	ff	f1	ab	a5	b7	b9	93	9d	8f	81
3	3b	35	27	29	03	0d	1f	11	4b	45	57	59	73	7d	6f	61
4	ad	a3	b1	bf	95	9b	89	87	dd	d3	c1	cf	e5	eb	f9	f7
5	4d	43	51	5f	75	7b	69	67	3d	33	21	2f	05	0b	19	17
6	76	78	6a	64	4e	40	52	5c	06	08	1a	14	3e	30	22	2c
7	96	98	8a	84	ae	a0	b2	bc	e6	e8	fa	f4	de	d0	c2	cc
8	41	4f	5d	53	79	77	65	6b	31	3f	2d	23	09	07	15	1b
9	a1	af	bd	b3	99	97	85	8b	d1	df	cd	c3	e9	e7	f5	fb
A	9a	94	86	88	a2	ac	be	b0	ea	e4	f6	f8	d2	dc	ce	c0
B	7a	74	66	68	42	4c	5e	50	0a	04	16	18	32	3c	2e	20
C	ec	e2	f0	fe	d4	da	c8	c6	9c	92	80	8e	a4	aa	b8	b6
D	0c	02	10	1e	34	3a	28	26	7c	72	60	6e	44	4a	58	56
E	37	39	2b	25	0f	01	13	1d	47	49	5b	55	7f	71	63	6d
F	d7	d9	cb	c5	ef	e1	f3	fd	a7	a9	bb	b5	9f	91	83	8d

Tab. 3.6: Vyhledávací tabulka pro násobení 14

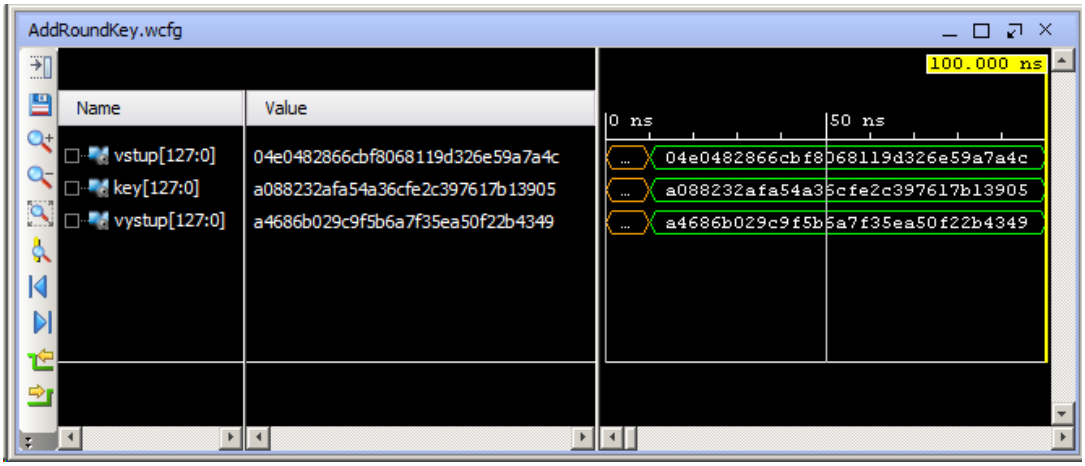
3.4 AddRoundKey

Poslední částí je přičtení šifrovacího klíče. Tento proces je proveden pomocí exkluzivní disjunkce. Vstupní blok dat je stejně velký jako šifrovací klíč, tedy 128 bitů. Obě tyto hodnoty jsou známé, takže nezbývá než provést logickou operaci XOR.



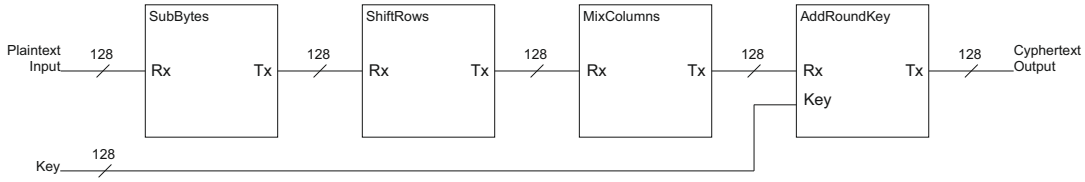
Obr. 3.10: Blokové schéma AddRoundKey

```
RX  <= x"04E0482866CBF8068119D326E59A7A4C";
KEY <= x"A088232AFA54A36CFE2C397617B13905";

architecture structural of AddRoundKey is
begin
  TX <= RX xor KEY;
end structural;
```



Obr. 3.11: Praktická ukázka AddRoundKey

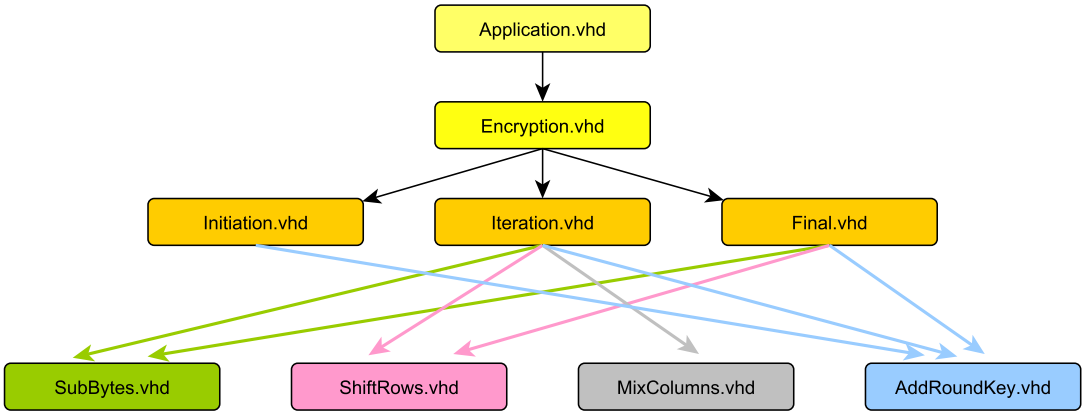
Jestliže spojíme všechny bloky za sebe, vznikne nám jedna runda, neboli iterace. Toto spojení je zobrazeno na blokovém schéma 3.12.



Obr. 3.12: Blokové schéma jedné rundy

3.5 Šifrování a dešifrování ECB

V předchozích kapitolách byla implementována jedna runda, potřebná ke správnému šifrování AES. Nyní je třeba vytvořit ucelený projekt, který bude reprezentovat šifrování a dešifrování AES v módu ECB. Struktura mezi jednotlivými entitami, které tvoří blok AES, je hierarchická. Jednotlivé komponenty jsou mezi sebou provázány. Hierarchickou strukturu popisuje obrázek 3.13. V nejnižší vrstvě jsou základní bloky algoritmu AES. Nad nimi jsou tři komponenty, které reprezentují tři dílčí šifry, popsané v teoretické části 1.1. Komponenta `Encryption` (jestliže mluvíme o šifrování) obsahuje všechny potřebné podkomponenty, ze kterých skládá kompletní šifrování za pomocí 128 bitového klíče. V nejvyšší pozici struktury je komponenta `Application`, která je provázaná s nižšími vrstvy, ale především komunikuje s celým frameworkem NetCOPE. Komponenta `Application` se odkazuje na aplikační jádro frameworku a komunikuje se všemi periferiemi potřebnými ke správné funkčnosti karty.

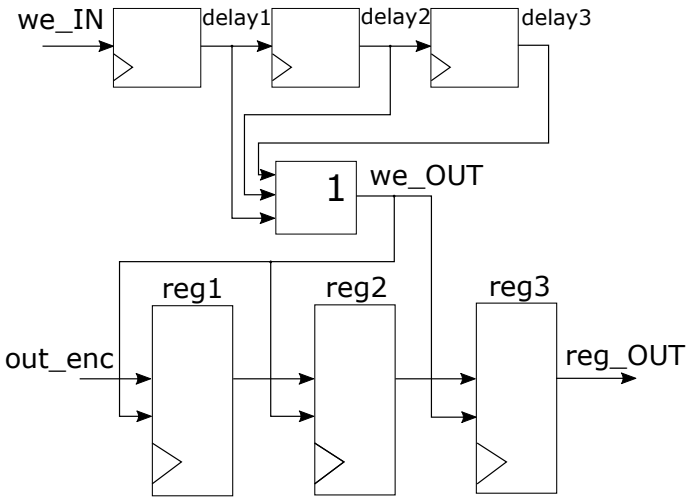


Obr. 3.13: Hierarchická struktura komponent

V samotné architektuře komponenty Application jsou definovány pomocné signály a hlavně jsou zde namapovány fyzické registry, které jsou na kartě. Každý registr je velký 128 bitů, což vystačí na uchování informace všech bloků dat. Jakým způsobem se s registry pracuje, jak se k nim přistupuje atd. je popsáno v kapitole Testování na kartě 4.3.

V adresovém prostoru protokolu MI32 (zmiňováno v kap. 2.4.1) jsou vytvořeny registry pro uložení klíčů (reg_KEY), dále vstupní registr (reg_IN) a výstupní registr (reg_OUT). Do nich jsou postupně nahrávána data, která chceme šifrovat. Při první implementaci bylo zjištěno, že výpočet AES při taktu 100 MHz neproběhne v jednom hodinovém cyklu. Bylo na snaze obdržet na výstupu správné hodnoty bez toho, aby uživatel musel výpočet spouštět vícekrát za sebou. Varianty, jak tento problém eliminovat, jsou dvě. Buď se sníží takt hlavního hodinového signálu, tím se sníží výkon karty a výpočet se provede v jednom taktu. Tento způsob zjištění správné hodnoty frekvence tzv. metodou pokus-omyl je zdlouhavý, protože se při každé změně kmitočtu musí spustit nová syntéza.

Druhá varianta se zdála být efektivnější. Řešení spočívá v tom, že se uměle pomocí vhodně umístěných registrů signál opozdí a do výstupního registru (reg_OUT) se zapíše již správná hodnota. Přesný počet registrů, které bylo do systému potřeba vložit se zjistilo snáze, než v předchozí variantě. Na obrázku 3.14 je blokové schéma zapojení registrů do kaskády. Při každém hodinovém taktu se hodnota registru přepisuje a vše řídí povolovací signál pro zápis do registru `we_IN`. Při zpětném zkoumání se zjistilo, že již v registru č. 2 byla uložena správná hodnota. Z toho je patrné, že zpoždění bylo 2 hodinové takty, proto do obvodu stačilo vložit 2 registry. Ty zpozdily signál pro zápis a ve výstupním registru `reg_OUT` byla uložena správná hodnota.



Obr. 3.14: Blokové schéma zpoždění výpočtu

Ve zdrojovém kódu byl tento problém vyřešen následovně. Byl vytvořen pomocný signál `enc_OUT`, který drží informaci z výstupu AES bloku. Předá jej do registru `reg_OUT` vždy o 2 zpožděné takty oproti signálu vstupnímu. Při taktu 100 MHz je se zpožděním doba vybavení 20 ns.

```
if (RISING_EDGE(mi_clk_i)) then
    if (we_IN(i) = '1') then
```

```
        reg_IN(31+i*32 downto i*32) <= MI_DWR;
      end if;
      if (we_OUT = '1') then
        reg_OUT(31+i*32 downto i*32) <= enc_OUT(31+i*32 downto i*32);
      end if;
  end if;


  if (RISING_EDGE(mi_clk_i)) then
    we_exp0 <= we_IN(0);
    we_exp1 <= we_exp0;
    we_OUT <= we_exp1;
  end if;


  encrypt_i: entity work.encryption
    port map (RX => reg_IN,
              TX => enc_OUT);
```

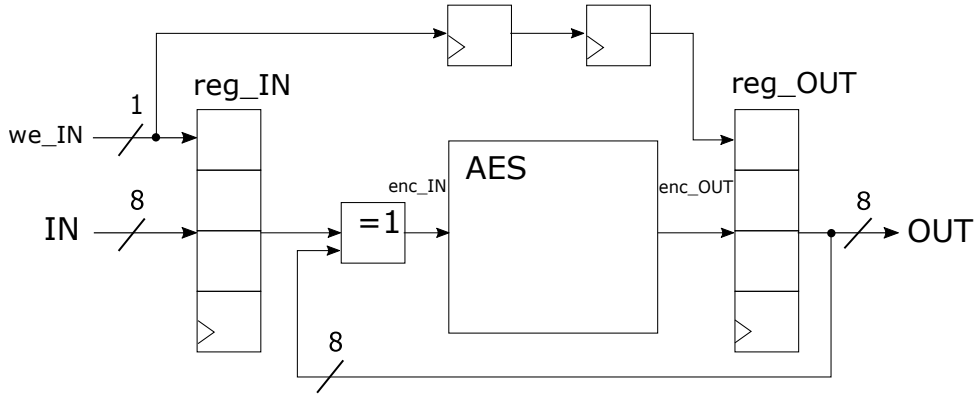
Nyní stačí nahrát data do vstupního registru reg_IN a na výstupu reg_OUT obdržíme zašifrovaný text. Data se šifrují jednotlivě po blocích, snadnou obsluhu zajistí vytvořený program, který nahraje data ze vstupního souboru a následně výstup uloží do nového souboru. Dešifrování je zajištěno inverzní operací. Provádí jej firmware určený pro dešifrování ECB.

3.6 Šifrování a dešifrování CBC

Podobně jako provozní režim ECB, tak i tento mód CBC vychází ze stejné hierarchické struktury zobrazené na obrázku č. 3.13. Liší se pouze v komponentě `Application`. Ta se nejen jinak jmenuje, nyní Application_AES_CBC_Encryption nebo Application_AES_ECB_Decryption, ale pracuje i odlišně se vstupními a výstupními signály. Fyzické vstupní a výstupní registry jsou zachovány, stejně jako registry pro uložení klíčů.

Hlavní změna je v části, která pracuje se vstupním signálem z registru reg_IN a výstupním signálem reg_OUT. Architektura využívá pomocné registry enc_IN a enc_OUT, do kterých přivádí signály pro samostatný blok, vykonávající operaci šifrování AES. Tento popis je zřejmý z blokového schéma 3.15.

Inicializační vektor, který je potřebný pro správnou funkci CBC je interpretován před začátkem šifrování ve výstupním registru reg_OUT. Změna je taková, že registr je zpřístupněn i pro zápis. Toho se využije na počátku, kdy se pomocí programu (nebo ručně) uloží inicializační vektor do registru OUT. Jakmile se nahrají data do vstupního registru reg_IN spustí se šifrování. V prvním kroku se do signálu enc_IN



Obr. 3.15: Blokové schéma AES CBC

uloží výsledek exkluzivního logického součetu signálů IN a OUT. Tento signál je přiveden na vstup bloku AES a výsledek se po dvou hodinových taktech uloží do výstupního registru reg_OUT. Operace je prováděna znovu s tím, že výstup je přiveden ke vstupu nového datového bloku, kde je opět „xorován“.

U dešifrování je používán blok AES s inverzními operacemi, ale struktura aplikačního jádra je zachována. Hlavním rozdílem je pozice, kde se provádí operace XOR s předchozí hodnotou. U šifrování se „xorování“ provádělo před vstupem do bloku AES, nyní je operace provedena na výstupu dešifrovacího procesu. Aby byla zachována správná hodnota předchozího bloku dat, je třeba signál uložit do mezi-registru, který informaci předá novému bloku. Tento registr označen reg_IV obsahuje stejnou informaci jako signál reg_IN, pouze je o jeden takt zpožděn, aby nedocházelo k přepisování signálu v jednom cyklu. Do výstupního registru reg_OUT je opět uložena správná hodnota.

4 TESTOVÁNÍ

V této kapitole jsou shrnuty všechny výsledky implementace, které byly ověřeny pomocí simulace nebo přímo fyzicky na síťové kartě. Zaměříme se na správnost výsledků při simulaci, které vychází z teoretické části algoritmu AES. U testování na síťové kartě Combo je důležité se zaměřit na rychlost výpočtu a možnosti práce s daty. Kvůli těmto různým druhům testování je kapitola rozdělena na tři části.

4.1 Simulace

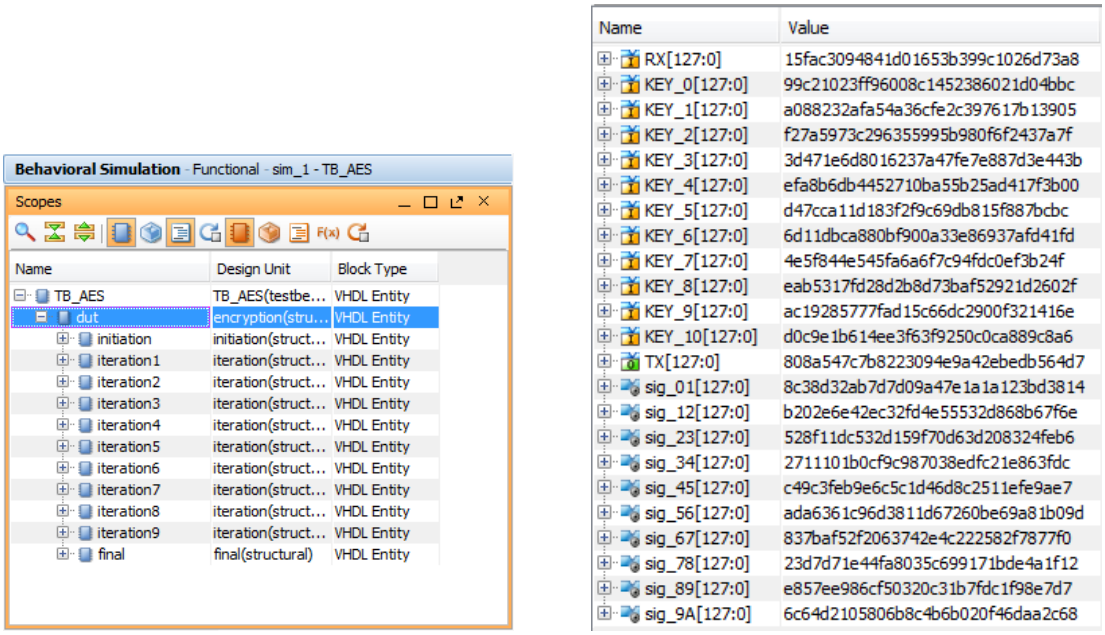
Pro jednodušší návrh určité problematiky je využíván proces simulace, kdy veškeré potřebné informace jsou dostupné podle potřeb programátorovi během krátké doby. Ten je schopen díky mezivýsledkům označit dosud provedené kroky za správné, případně odladit rychle a efektivně chyby, kterých se doposud dopustil. Tímto způsobem bylo realizováno šifrování AES v prostředí Vivado®, které je popsáno v kapitole 2.2. Simulovány jsou jednotlivé bloky transformací, uvedené v teoretické části 1.1 a následně navrženy v kapitole 3.

Cílem je simulovat testovanou entitu pro všechny možné kombinace a snímat výstupní signály. Pro ověření funkčnosti je třeba využít tzv. testbench. Jedná se o kód napsaný za účelem ověření původního zdrojového VHDL kódu. Obvykle nemá žádné vstupy a výstupy. Má jen namapované entity, které chceme ověřit a jednotlivým signálům jsou přiřazeny vstupní hodnoty.

Za pomoci vytvořených testbenchů pro jednotlivé komponenty se odsimulovaly uvedené kroky algoritmu. Jedná se o zdrojové kódy TB_SubBytes.vhd, TB_ShiftRows.vhd, TB_MixColumns.vhd a TB_AddRoundKey.vhd. V nich jsou vytvořeny potřebné signály se simulovanými daty, které jsou předány požadované entitě. Zde je uveden příklad testbenche pro transformaci SubBytes.

```
entity TB_SubBytes is
end entity TB_SubBytes;

architecture testbench of TB_SubBytes is
  signal vstup, vystup : std_logic_vector(127 downto 0);
  begin

  test : process
  begin
    wait for 10 ns;
    vstup <= x"19A09AE9_3DF4C6F8_E3E28D48_BE2B2A08";
  end process test;
end architecture testbench;
```

Následně se sjednotily všechny komponenty do jednoho projektu Vivada a provedla se finální simulace. Ta představuje implementaci a ověření výsledků celého procesu šifrování. Testbench je pojmenován TB_AES.vhd. V něm je namapována hlavní entita encryption, která vykonává celé šifrování AES. Jsou zde vytvořeny signály RX (vstup) a TX (výstup). Dále jsou procesu předány všechny potřebné klíče pro jednotlivé rundy. Za pomoci vytvořených testbenchů pro jednotlivé komponenty se odsimulovaly uvedené kroky algoritmu. Na snímku 4.1 jsou vidět mezikroky výpočtu, v pravé části jsou zobrazeny signály označující vstup, výstup šifrování a jednotlivé signály. Na obrázku 4.2 je prezentována simulace v programu Vivado.

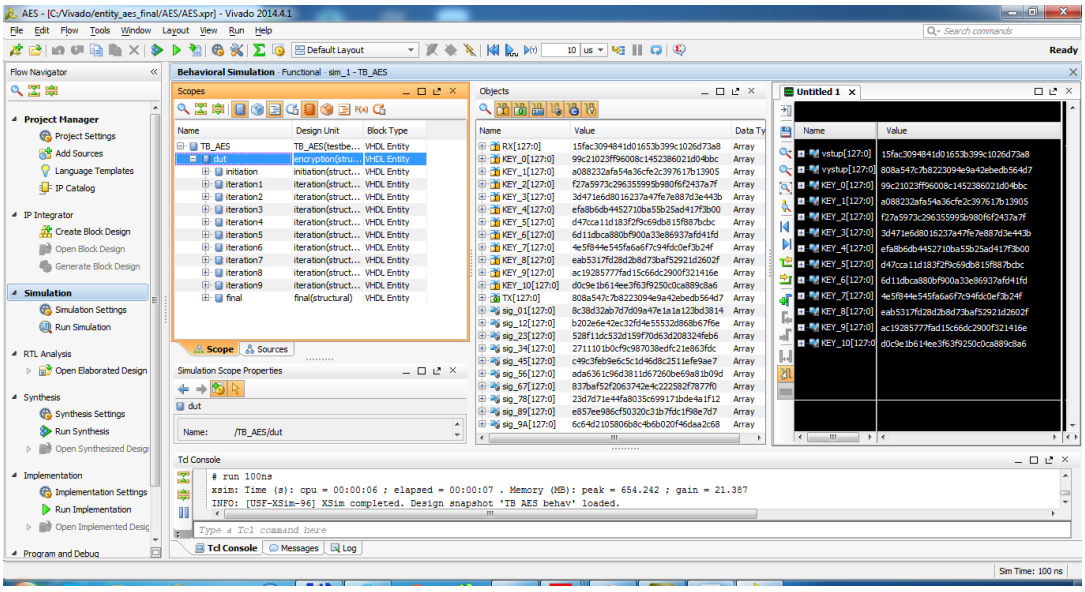


Obr. 4.1: Ukázka simulace mezikroků

4.2 Syntéza

Vývoj je prováděn na syntézním serveru, kde za pomocí vývojového softwaru Vivado je vytvářen firmware pro síťovou kartu. Na serveru se používá 64 bitový operační systém Linux CentOS release 6.6 (Final). Server je osazen čtyřjádrovým procesorem Intel® Xeon® E5-2643 (3.30 GHz) s operační pamětí 6 GB. Pro vývoj je použit nejnovější NetCOPE verze 8.02.02 z května roku 2015. Je vhodně navržen pro práci s kartou COMBO-80G. Vivado na syntézním serveru je verze 2013.4 z prosince roku 2013.

Kvůli více jak ročnímu rozdílu verzí frameworku a syntéznímu softwaru je nutné při spuštění terminálu na syntézním serveru vždy zadat následující příkaz. Ten namapuje všechny cesty k Vivado potřebné pro správnou syntézu.



Obr. 4.2: Ukázka simulace v programu Vivado

```
[xsmeka11@LFPGA ~]$ source netcope80g.sh
```

Na serveru je třeba mít umístěný ResourceCD, jak je označován výchozí adresář NetCOPE. Změníme aktivní adresář projektu a do složky `core` umístíme naše zdrojové kódy popisující šifrování. Všechny tyto soubory vhd je třeba specifikovat v souboru `Modules.tcl` nacházející se v adresáři `src`.

```
[xsmeka11@LFPGA ~]$ cd NetCOPE_8.02.02_ResourceCD/applications/
                       nic_10g8/src/

[xsmeka11@LFPGA src]$ ls
config          core                make.exe   Modules.tcl   Vivado.tcl
constraints     load_xil_env.bat    Makefile   tcl
```

Příkazem `make` se spustí syntéza. Jednotlové její kroky jsou zobrazeny na obrázku č. 2.2.

```
[xsmeka11@LFPGA src]$ make
```

Po zhruba dvou hodinách je syntéza u konce. Přepneme se do adresáře `build`, kde je vygenerovaný soubor `fpga.bit`

```
[xsmeka11@LFPGA ~]$ cd NetCOPE_8.02.02_ResourceCD/applications/
                       nic_10g8/build

[xsmeka11@LFPGA build]$ ls
fpga.bit  work
```

Do FPGA karty se nahrává firmware ve formátu `.mcs`, který získáme spuštěním skriptu `bit2mcs.sh` následujícího formátu.

```
#!/bin/sh
promgen -p mcs -data_width 16 -w -u 0 fpga.bit -o fpga.mcs
```

Nyní máme vytvořený firmware `fpga.mcs` obsahující šifrování AES potřebný pro kartu Combo.

```
[xsmeka11@LFPGA build]$ ls
bit2mcs.sh  fpga.bit  fpga.cfi  fpga.mcs  fpga.prm  work
```

4.3 Testování na kartě

Testování probíhá na serveru, kde je umístěna síťová karta Combo, do které je nahrán firmware. Ten obsahuje framework NetCOPE s navrženou implementací AES, kdy lze na fyzickém hardwarovém zařízení testovat věrohodnost a rychlost šifrování bloků dat.

Na serveru se síťovou kartou Combo se používá 64 bitový operační systém Linux CentOS 6.2 Final (NetCOPE 05-0C.1). Server je osazen šestijádrovým procesorem Intel® Xeon® E5-2620 (2.00 GHz) s operační pamětí 16 GB.

V prvním kroku je třeba zkontrolovat, zda karta Combo má načteny všechny potřebné ovladače.

```
[root@localhost ~]# lsmod | grep szedata

szedata2_cv3
szedata2
combov3
combo6core
```

V dalšík kroku je možné nahrát pomocí příkazu `csboot` vytvořený firmware, který chceme testovat. Během pár minut dojde k nabootování karty.

```
[root@localhost ~]# csboot -f 100 encryption-AES.mcs
```

Jestli se firmware úspěšně na kartu nahrál ověříme pomocí příkazu `csid -s`, který nám vypíše veškeré informace o firmwaru.

```
[root@localhost ~]# csid -s

Built at : 2015/05/11 08:52:37
Board    : COMBO-80G
Subtype  : VXT690
S/N      :
Speedgr. : 2
```

```
Addon0    : n/a
Chip0     : n/a
S/N0      :
Addon1    : n/a
Chip1     : n/a
S/N1      :
Channels  : 8/8 (RX/TX)
Firmware  : ok
SW        : 0x41c10700
HW        : 0x00010000
Text      : NETCOPE-NIC_10G8
Caps      : 0x00000000
ID ver.   : 0x0104
NetCope   : 0x0802

Driver [combov2] szedata2cv3: active
  (0x41c10700-0x41c10800) {}
  (0xa41c0300-0xa41c0400) {}
  (0x5d010000-0x5d011000) {}
```

Nahrání proběhlo v pořádku a nyní je možné zkontrolovat vstupní a výstupní registr, který by měl obsahovat samé nuly, jelikož jsme na vstupu nepřidělily žádná data, která by měl šifrovat. To ověříme následujícím příkazem. Je důležité zadat správné adresy, ze kterých chceme číst.

```
[root@localhost ]# csbus -c 4 0x2000000
  00000000 00000000 00000000 00000000

[root@localhost ]# csbus -c 4 0x2000010
  00000000 00000000 00000000 00000000
```

První adresa 0x2000000 odpovídá vstupnímu registru a druhá 0x2000010 adresa je registr výstupní. Nyní je možné do registru zapsat data, která chceme zašifrovat. Po naplnění celého registru, se zašifrovaná data uloží do registru výstupního.

Pro jednodušší obsluhu šifrování je vytvořen program, který zapisuje vstupní data ze souboru do registrů na síťové kartě a naopak vyčítá šifrový text z karty a ukládá jej do výstupního souboru. Vstupní data musí být uložena po blocích 128 bitů v hexadecimálním tvaru (tedy 32 znaků), na každém řádku jeden blok dat. Program postupně nahrává řádek po řádku na síťovou kartu, kde probíhá výpočet a zároveň ukládá zašifrované bloky do výstupního souboru. Uživatel pak snáze výstupní soubor překontroluje, zda šifrování probíhá správně. Tento program byl zmíněn v předchozím textu v části 2.4.1. Po kompilaci se spouští na serveru s kartou následujícím příkazem.

```
[root@localhost ]# make
```

```
  cc encryption-AES.c -lcombo -std=c99 -D_BSD_SOURCE -o component

[root@localhost ]# ./encryption-AES
RX
  328831e0 435a3137 f6309807 a88da234
TX
  3902dc19 25dc116a 8409850b 1dfb9732
```

```
[root@localhost ]# csbus -c 4 0x2000000
  328831e0 435a3137 f6309807 a88da234

[root@localhost ]# csbus -c 4 0x2000010
  3902dc19 25dc116a 8409850b 1dfb9732
```

Testování proběhlo v pořádku, výstupní data odpovídají správnému výsledku, který byl očekáván. Stejným způsobem lze testovat ostatní firmwary, které jsou vytvořeny. Jsou implementovány celkem 4 projekty. Dva z nich jsou šifrování a dešifrování ECB módu AES algoritmu a zbylé dva popisují šifrování a dešifrování módu CBC.

Pro snadnější postup testování jsou v příloze připojeny video-návody, které zaznamenávají kroky, jak postupovat při simulaci, syntéze nebo práci s kartou. U simulace je použit software Vivado a v něm spuštěna simulace šifrovacího procesu. U syntézy se popisuje její spuštění a následné vytvoření firmwaru. Poslední video obeznámí uživatele jakým způsobem nahrát testovaný firmware na kartu a jak otestovat jeho správnou funkčnost.

5 ZÁVĚR

Cílem diplomové práce bylo nastudovat jazyk VHDL, princip FPGA karet a šifrování obecně. Bylo nutné si zjistit potřebné informace, jak šifrovací algoritmus AES pracuje a jaké jsou možnosti implementace na FPGA. Získat patřičné znalosti ohledně programovacího jazyka, jeho syntaxi a způsob práce se vstupními signály. Osvojit si práci ve vývojovém prostředí Vivado a pochopit funkci simulace a následné syntézy.

Jak je uvedeno v předchozích kapitolách, byly naprogramovány čtyři základní bloky algoritmu a otestována jejich správná funkčnost. Věrohodnost výstupu byla ověřena pomocí simulace. Výsledný návrh šifrování se zaměřuje pouze na správnost algoritmu s použitím operačního módu ECB a CBC. Pro obě varianty jsou v příloze 2 různé projekty. Bližší uspořádání projektů a souborů je popsáno v příloze.

Současný návrh neřeší práci s klíči a expanzi klíče, ty jsou pevně definovány ve zdrojovém kódu komponenty Application. Klíčů je celkem 11, pro jednoduchost pracujeme pouze s klíčem 128 bitů. Klíče jsou definovány jako signály pod označením KEY_0, KEY_1, ..., KEY_10. Určení právě těchto klíčů vychází z algoritmu pro expanzi klíče. Správnost je možné ověřit výpočtem nebo využitím různých softwarových aplikací.

Pro dešifrování je vždy vytvořen nový projekt, který se zaměřuje na inverzní operace. Správnost lze simulovat v prostředí Vivado® nebo testovat na síťové kartě za pomocí dešifrovacího firmwaru. Pro testování je třeba použít správná data, aby bylo možné ověřit správnou funkčnost šifrování i dešifrování.

Rychlost šifrování dat je závislá na frekvenci jádra samotné karty. V našem případě je takt nastaven na 100 MHz. Šifrování se však o jeden hodinový takt zpožděno, proto uvažujeme s dobou výpočtu 20 ns. Za tuto dobu je zpracován jeden blok dat 128 bitů. Z toho je možné určit teoretickou rychlost zpracování. Jestliže zpracujeme za 20 ns 128 bitů dat, propustnost šifrování je 6,4 Gb/s. Kdyby se operace provedla v jednom hodinovém taktu při 100 MHz, rychlost by vzrostla na 12,8 Gb/s.

LITERATURA

[1] PINKER, Jiří, POUPA, Martin. *Číslicové systémy a jazyk VHDL*. 1. vyd. Praha: BEN - technická literatura, 2006, 349 s. ISBN 80-730-0198-5.

[2] BURDA, Karel. *Aplikovaná kryptografie*. 1. vyd. Praha: VUTIUM, 2013, 255 s. ISBN 978-80-214-4612-0.

[3] FIPS-197. ADVANCED ENCRYPTION STANDARD (AES). USA: NIST, 2001. Dostupné z: `http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf`

[4] COMBO-80G FPGA karta. [online]. [cit. 2015-04-10]. Dostupné z: `https://www.invea.com/cs/produkty-sluzby/fpga-karty/combo-80g`

[5] Vývojový framework NetCOPE. [online]. [cit. 2015-04-10]. Dostupné z: `https://www.invea.com/cs/produkty-sluzby/fpga-development-kit/netcope`

[6] Struktura obvodu typu FPGA. [online]. [cit. 2014-11-15]. Dostupné z: `http://www.abclinuxu.cz/blog/digital_design/2013/1/programovatelna-logika-ii-fpga`

[7] Vivado Design Suite User Guide: Implementation. [online]. [cit. 2014-11-27]. Dostupné z: `http://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_4/ug904-vivado-implementation.pdf`

SEZNAM SYMBOLŮ, VELIČIN A ZKRATEK

AES	Advanced Encryption Standard
bit	základní datová jednotka
bajt	jednotka množství dat, osm bitů
C	blok šifrového textu
CBC	mód Cipher Block Chaining
CentOS	distribuce operačního systému Linux
CLB	Configurable Logic Block
CFB	mód Cipher Feedback
CTR	mód Counter
CWDM	Coarse Wavelength Division Multiplex
D	dešifrovací transformace
E	šifrovací transformace
ECB	mód Electronic Codebook
FLU	FrameLinkUnaligned
FPGA	Field Programmable Gate Array
h	označení hexadecimálního slova
IOB	Input/Output Block
K	šifrovací klíč
NetCOPE	konfigurovatelný framework
OFB	mód Output Feedback
QSFP	Quad Small Form-Factor Pluggable
Rx	Receiver – vstup bloku
SSH	Secure Shell
Tx	Transmitter – výstup bloku

V	vstupní blok bitů
VHDL	Very High Speed Integrated Circuit Hardware Description Language
VNC	Virtual Network Computing
W	výstupní blok bitů
XOR	exkluzivní logický součet
X	význam zkratky
XTS	Xor–encrypt–xor–based tweaked–codebook mode with ciphertext stealing
Z	vstupní blok

A OBSAH PŘILOŽENÉHO DVD

Pro snadnější orientaci v adresářích a souborech na přiloženém DVD je zde uveden seznam. Popis je uváděn z kořenového adresáře DVD (symbol /).

- /firmware/
  - složky s vytvořenými firmwary a soubory pro kartu Combo–80G
- /firmware/AES-CBC/
  - firmwary pro šifrování/dešifrování AES CBC mód, program pro obsluhu
- /firmware/AES-ECB/
  - firmwary pro šifrování/dešifrování AES ECB mód, program pro obsluhu
- /simulation code/
  - složky s projekty pro simulaci
- /simulation/encryption/
  - spustitelná simulace v programu Vivado – projekt šifrování
- /simulation/decryption/
  - spustitelná simulace v programu Vivado – projekt dešifrování
- /source code/
  - složky se zdrojovými kódy v jazyce VHDL
- /source code/encryption/
  - zdrojové kódy v jazyce VHDL – projekt šifrování
- /source code/decryption/
  - zdrojové kódy v jazyce VHDL – projekt dešifrování
- /synthesis/
  - složky s upravenými soubory NetCOPE potřebné pro syntézu, stačí nakopírovat do adresáře NetCOPE
- /synthesis/AES-CBC-DEC/
  - soubory potřebné pro dešifrování (adresář core)
- /synthesis/AES-CBC-ENC/
  - soubory potřebné pro šifrování (adresář core)
- /synthesis/AES-ECB-DEC/
  - soubory potřebné pro dešifrování (adresář core)
- /synthesis/AES-ECB-ENC/
  - soubory potřebné pro šifrování (adresář core)
- /text/
  - obsahuje textovou část diplomové práce ve formátu PDF
- /video/
  - obsahuje videonávody pro spuštění simulace a syntézy, pro obsluhu karty (nahrání firmwaru, testování šifrování)